UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE CIENCIAS

PLONEVOTECRYPTOLIB: UNA BIBLIOTECA CRIPTOGRÁFICA PARA LA IMPLEMENTACIÓN DE ELECCIONES EN LÍNEA SECRETAS Y VERIFICABLES POR ELECTORES

T E S I S

QUE PARA OBTENER EL TÍTULO DE:

LICENCIADO EN CIENCIAS DE LA COMPUTACIÓN

P R E S E N T A :

LÁZARO CLAPP JIMENÉZ LABORA

DIRECTOR DE TESIS:

DR. SERGIO RAJSBAUM GORODEZKY

2011

Hoja de Datos del Jurado

1. Datos del alumno
Clapp
Jiménez Labora
Lázaro
55 73 98 58
Universidad Nacional Autónoma de México
Facultad de Ciencias
Ciencias de la Computación
304684029

2. Datos del tutor
Dr
Sergio
Rajsbaum
Gorodezky

3. Datos del sinodal 1
Dr
Nils Heye
Ackermann

4. Datos del sinodal 2
Dr
José de Jesús
Galaviz
Casas

5. Datos del sinodal 3
Dra
Elisa
Viso
Gurovich

6. Datos del sinodal 4
Dr
Carlos
Velarde
Velázquez

7. Datos del trabajo escrito
PloneVoteCryptoLib: Una biblioteca criptográfica para la implementación de elecciones en línea secretas y verificables por electores.
128p
2011

Agradecimientos

Agradezco a mi madre por su apoyo y sabiduría, durante toda mi vida, pero, en especial, durante los cuatro años de mis estudios de licenciatura y los seis meses de mi trabajo de tesis (y demás trabajos directa o indirectamente relacionados). Le agradezco sus consejos y recomendaciones cuando me desesperaba escribiendo el presente documento; trabajo que a veces parecía inaceptable me tomara más de dos meses y otras parecía imposible de terminar en doce. Le agradezco las incontables veces que me ayudó a entender una demostración, buscar una referencia o aprender como plasmar cierto hecho dentro de un trabajo académico. Así pues, le agradezco por mostrarme el gran valor de tener una matemática en casa.

Agradezco a mi asesor, el Dr. Sergio Rajsbaum, por su apoyo en la definición y realización de este trabajo. Agradezco su paciencia y comprensión, cuando, tras haberme asignado completar un proyecto ya hecho, decidí en cambio construir desde cero mi propia solución al problema, proporcionando las garantías de seguridad que a mí me parecieron necesarias. Agradezco su apoyo e insistente recomendación al impulsarme a presentar este trabajo en Plone Symposium East 2011, así como su ayuda y consejos sobre cómo mejor prepararme para comunicar mis contribuciones actuales y mi visión para el proyecto. En sus esfuerzos por ayudarme, 45 diapositivas fueron sacrificadas, junto con mi calma y tranquilidad al tener que remplazarlas dos días antes de emprender el viaje. Sin embargo, gracias a ello, mi presentación pasó de ser una confusa y vertiginosa carrera de detalles a una precisa y mesurada herramienta para atraer interés hacia PloneVote. Le debo, completamente, el éxito que tuvo mi plática. Le agradezco finalmente, mas no en menor medida, dos de los mejores y más profundamente apasionantes cursos que tomé durante la carrera: Procesos Paralelos y Complejidad Computacional.

Agradezco al Dr. Nils Ackermann su apoyo durante toda la duración del presente trabajo: agradezco su ayuda desde antes de que comenzara el proyecto, al buscar y proporcionarme información, enlaces y referencias sobre sistemas seguros de elecciones en línea; sus consejos y recomendaciones a lo largo de la realización del proyecto; y, al final de éste, en su papel como sinodal, sus valiosos comentarios que contribuyeron a la precisión matemática de este documento.

Agradezco a la Dra. Elisa Viso y al Dr. José Galaviz haber aceptado ser sinodales de mi examen profesional y sus correcciones y consejos respecto a este trabajo. Mucho más aún, agradezco todo lo que aprendí de ellos. Sus excelentes cursos de Matemáticas Discretas (Elisa), Introducción a las Ciencias de la Computación I y II (José) y Teoría de la Computación (Elisa), fueron para mí, como para muchos otros estudiantes de la carrera, la base de mi formación y entendimiento como computólogo, por lo que les estaré eternamente agradecido. Lejos de disminuir en calidad, los cursos avanzados que tomé con ambos: un Seminario de Programación (José) y el curso de Compiladores (Elisa), fueron, si es que tal cosa es posible, aún mejores y

más apasionantes que sus cursos introductorios.

Agradezco al Dr. Carlos Velarde aceptar ser sinodal de mi examen profesional, sin siquiera conocerme previamente, y sus dedicadas y completas correcciones al borrador de este trabajo.

Agradezco al Dr. Jorge Urrutia sus extraordinarios cursos de algoritmos, los cuales figuran, sin lugar a duda alguna, entre los más estimulantes y satisfactorios de mi carrera.

Agradezco al Dr. Favio Miranda sus excelentes cursos de lenguajes de programación, los cuales tuvieron un papel importante en mi elección de sistemas de programación como área de estudio para mi posgrado; aun si el aspecto práctico de mi elección le puede resultar sacrílego al estimado Dr. Favio.

Agradezco, amplia y profundamente, a todos mis profesores por los conocimientos que me trasmitieron a lo largo de la carrera, y por su compromiso y dedicación.

Agradezco a la Facultad de Ciencias y a la Universidad Nacional Autónoma de México por haberme dado acceso a tan excelentes estudios. Agradezco al Instituto de Matemáticas por haberme otorgado una beca de lugar para realizar mi trabajo de tesis. Agradezco a México, como nación, por hacer posible un lugar como la UNAM.

Este trabajo fue parcialmente apoyado por el proyecto PAPIIT IN123109 "Sistemas Distribuidos de Información".

Índice general

Capítulo 1

Introducción

1.1. Objetivos de este trabajo

Este trabajo presenta la biblioteca PloneVoteCryptoLib, la cual forma el núcleo criptográfico del sistema PloneVote, actualmente en desarrollo.

PloneVote es un sistema para soporte de elecciones en línea seguras y verificables por electores, diseñado específicamente para elecciones intra-institucionales. El objetivo principal de PloneVote es su uso como un sistema para la realización de elecciones internas dentro del Instituto de Matemáticas de la Universidad Nacional Autónoma de México. Con esto en mente, está planeado para integrarse fácilmente con la infraestructura de sistemas actualmente utilizada por tal institución. Es contemplado, sin embargo, que el sistema sea publicado como software libre y abierto a la comunidad internacional para su uso y evaluación.

PloneVote basa los aspectos de seguridad de su funcionamiento en el protocolo de Elecciones Simples Verificables, presentado por Josh Benaloh en [Benaloh2006]. Este protocolo depende, a su vez, de una serie de primitivos criptográficos complejos, en particular: cifrado de umbral y redes de mezcla verificables. Introduciremos en este trabajo la teoría en la que se basan tales primitivos. Presentaremos, simultáneamente, una implementación práctica de éstos, realizada como parte del presente trabajo: la biblioteca PloneVoteCryptoLib.

PloneVoteCryptoLib ha sido desarrollada como un componente real del sistema PloneVote. El proceso de análisis de requerimientos del sistema completo, así como su arquitectura general, fueron realizados como parte de este trabajo. PloneVoteCryptoLib representa la base fundamental del proyecto PloneVote y ha sido desarrollada a conciencia para permitir la construcción del sistema completo de elecciones en línea sobre ésta. Asimismo, se trata del componente más teóricamente interesante del sistema, y por tanto más adecuado a ser expuesto en un trabajo académico.

PloneVote es un sistema de software complejo, cuyo desarrollo completo e implementación queda fuera del alcance de este trabajo.

1.2. El problema de elecciones en línea

El método tradicional de llevar a cabo elecciones, en boletas de papel, es efectivo, sencillo de entender y, provistos controles administrativos adecuados, razonablemente confiable.

Por otra parte, el uso de sistemas remotos sobre Internet ha sido introducido en múltiples aspectos de la vida cotidiana (e.g. correo electrónico o banca en línea). En general, estos sistemas ofrecen ventajas de comodidad de uso importantes, como flexibilidad de horarios o la posibilidad de realizar acciones desde un equipo de cómputo en cualquier lugar del mundo. Es razonable suponer, entonces, que sistemas similares pueden ser utilizados para la realización de elecciones, ofreciendo ventajas análogas.

Un sistema para elecciones en línea no solo hace el proceso más cómodo para el elector, evitando que éste tenga que desplazarse a un lugar determinado para votar, dentro de horarios específicos. Además, puede facilitar la logística requerida de parte de las autoridades que conducen la elección, particularmente si existen múltiples estaciones de votación en lugares distintos y los votos deben ser transportados físicamente a una ubicación central para su conteo, como ocurre en el caso del Instituto de Matemáticas y sus unidades foráneas.

Es importante, sin embargo, cuidar que cualquier sistema electrónico usado para reemplazar un esquema de votación tradicional provea, al menos, un nivel de seguridad similar al del sistema que reemplaza. Más aún cuando, por regla general, ataques efectivos contra sistemas informáticos tienen el potencial de ser más difíciles de detectar y de alcance más amplio, comparados con ataques contra sistemas físicos tradicionales.

Han sido descritos en la literatura académica numerosos protocolos de votación seguros, basados en técnicas criptográficas. Algunos de estos protocolos no solo proporcionan las mismas garantías de seguridad que el método tradicional en papel, sino que añaden propiedades de seguridad adicionales, como verificabilidad por parte del elector ([Benaloh2006] es un ejemplo de tal protocolo).

Estos protocolos son, usualmente, más difíciles de entender que el método tradicional. Asimismo, implementarlos correctamente es una tarea considerablemente compleja. En particular, la descripción de las bases teóricas para [Benaloh2006], junto con nuestra implementación de sus operaciones principales, ocupará la mayor parte del presente trabajo.

Parte de los objetivos de un sistema de software para elecciones en línea seguras y verificables, consiste en abstraer la mayor complejidad posible de dicho sistema al elector, sin disminuir la seguridad del sistema o la confianza del elector en éste.

1.2.1. Propiedades de seguridad

Parte de lo que hace que la implementación de sistemas para elecciones sea una tarea compleja, es la multitud de propiedades de seguridad deseables para cualquier tal sistema, varias de ellas en aparente contradicción entre sí. Listamos a continuación las propiedades de seguridad que debería cumplir un esquema de votación ideal, independientemente de como éste sea implementado:

1. *Correctitud*: El resultado publicado para la elección debe ser congruente con los votos emitidos por los electores. Esto es, el resultado es un conteo preciso de los votos, y dichos votos realmente corresponden a las preferencias expresadas por los electores al momento de votar.

2. *Privacidad*: Debe ser imposible asociar ningún voto con la identidad del elector que lo emitió. En su sentido más fuerte, la propiedad de privacidad implica que, dado un elector cualquiera y el conjunto total de votos contados, cualquier voto tiene la misma probabilidad de haber sido emitido por dicho elector (para cualquier observador distinto del elector).

3. *Verificabilidad*: Debe ser posible para un elector, asegurarse de que los votos fueron contados correctamente y corresponden realmente a las preferencias del electorado.

   a) Un esquema es *individualmente verificable* si cada elector puede asegurarse que su voto fue capturado y contado correctamente.

   b) Un esquema es *universalmente verificable* si cualquier elector puede asegurarse que los votos de todos los electores fueron capturados y contados correctamente.

4. *Justicia*: Debe ser imposible para un elector obtener resultados parciales de la elección, correspondientes a los votos ya capturados, antes de emitir su propio voto. El objetivo de esta propiedad es evitar que un elector, votando estratégicamente, gane alguna ventaja o no, dependiendo del momento en que vote dentro del periodo de votación designado.

5. *Robustez*: El esquema de elección debe producir un resultado y mantener su funcionamiento adecuado bajo cualquier tipo de circunstancias. En particular, no debe ser posible eliminar votos ya capturados en el sistema, ni obstruir el conteo de los mismos o la verificación de una elección soportada por la propiedad de verificabilidad.

6. *Democracia*: Toda persona con derecho a votar en una elección debe ser capaz de emitir su voto de forma que éste sea contado. Asimismo, debe ser imposible para una persona sin derecho a votar en una elección particular, emitir un voto espurio.

7. *Incoercibilidad*: Nadie debe ser capaz de obligar a un elector a votar de cierta forma. Más aún, un elector no deberá tener manera de demostrar, a ninguna persona distinta de sí mismo, que votó de una manera especifica (si un elector pudiera mostrar cómo votó, esto permitiría la compra y venta de votos).

Es importante mencionar que ningún esquema de votación conocido, incluyendo el esquema tradicional en papel, satisface perfectamente, de forma simultánea, todas las propiedades anteriores. Un sistema para elecciones en línea será seguro en la medida en que ofrezca garantías sobre el cumplimiento de estas propiedades, adecuadas al escenario de uso esperado para dicho sistema.

1.2.2. Elecciones intra-institucionales

Durante el diseño de PloneVote y el desarrollo de su biblioteca criptográfica aquí presentada, nos enfocamos en el escenario de elecciones intra-institucionales, tomando las decisiones de seguridad más apropiadas para tal escenario. Una elección intra-institucional es aquella que ocurre dentro de una organización pública o privada de pequeño o mediano tamaño, tales como universidades, empresas y sus respectivas divisiones internas.

Existen diferencias importantes entre el escenario de elecciones intra-institucionales y escenarios más tradicionales en los cuales ha sido discutido el uso de sistemas de votación electrónica, tales como elecciones nacionales o estatales. PloneVote no está diseñado pensando en elecciones masivas y podría no ser apropiado, por razones tanto de seguridad como de escalabilidad, para soportar este tipo de elecciones. Se considera poco recomendable el uso de PloneVote para la realización de elecciones a nivel nacional, estatal o regional.

Una primera diferencia a tomar en cuenta entre ambos tipos de elecciones, es la cantidad de recursos esperados de un posible atacante. En una elección nacional es razonable imaginar un ataque generalizado a la infraestructura de cómputo pública, como parte de un intento de subvertir un proceso electoral. PloneVote permite a los electores votar desde cualquier equipo de cómputo personal. La mayoría de los sistemas operativos utilizados hoy en día en computadoras personales no cumplen estándares especialmente altos de seguridad (comparados con equipo electrónico especializado para votaciones, por ejemplo). En una elección nacional sería factible esperar ataques generalizados buscando subvertir un gran número de equipos de cómputo a nivel de su sistema operativo[1], con el objetivo de alterar su funcionamiento y comprometer el proceso de captura de votos. En el caso de elecciones intra-institucionales, la dificultad técnica de este tipo de ataque y el bajo

[1] Incluso, posiblemente, por parte de los fabricantes de tales sistemas operativos, sus distribuidores (OEM, cadenas de distribución, etc) o los desarrolladores de otros programas comúnmente utilizados sobre tales sistemas.

nivel de impacto de cualquier elección particular, hacen tal clase de ataques altamente improbable, más allá de la infraestructura de sistemas interna a la propia institución.

En segundo lugar, en elecciones intra-institucionales los electores se encuentran, en promedio, en una situación menos vulnerable ante coerción que la enfrentada por una amplia proporción de los electores en elecciones nacionales o estatales. Esto simplemente dado que las diferencias económicas, sociales y de poder en general, son menos marcadas entre electores y potenciales atacantes en el caso de una elección intra-institucional. PloneVote, como veremos más adelante, no ofrece protección significativa ante coerción en su modo de uso esperado. Esto podría constituir un problema de seguridad grave en el caso de elecciones masivas, pero lo consideramos una limitación aceptable para elecciones intra-institucionales.

Finalmente, nuestro sistema no necesitará escalar más allá de algunas decenas de miles de votos a lo más. Por su parte, sistemas diseñados para soportar elecciones nacionales o estatales deben usualmente ser capaces de manejar cientos de millones de votos[2].

1.3. Trabajo previo

Anteriormente al desarrollo de PloneVote, dos alternativas de sistemas para elecciones en línea han sido utilizadas por el Instituto de Matemáticas (IMATE), con resultados razonablemente satisfactorios. Discutiremos a continuación brevemente ambos sistemas, mencionando las limitaciones de éstos que llevaron a la concepción de PloneVote, así como las ventajas que éste busca ofrecer por sobre ambas alternativas.

1.3.1. ATVotaciones

ATVotaciones ([Zapata2008]) fue desarrollado también como un producto interno del IMATE, integrado con la infraestructura de sistemas del instituto, en particular con el manejador de contenidos Plone [Plone]. Al ser diseñado específicamente para el IMATE, este sistema se adecua, en términos de su interfaz y uso, a los requisitos de las elecciones llevadas a cabo en la institución.

Algunas características de este sistema fueron recuperadas en el diseño de PloneVote. Un ejemplo particular es la capacidad de utilizar metadatos sobre los usuarios del sistema Plone para generar automáticamente listas de electores y potenciales candidatos, de acuerdo con reglas administrativas.

Desafortunadamente, ATVotaciones presenta problemas serios de seguridad a un nivel fundamental de diseño, proveyendo tan sólo una protección

[2]Generalmente, esto implica sistemas altamente distribuidos. PloneVote, al menos en su versión inicial, está planeado para usar un solo servidor central.

marginal a las propiedades de seguridad vistas en 1.2.1, incluyendo correctitud y privacidad de la elección. Lo anterior es considerado inadecuado, particularmente comparado con el estado del arte en sistemas de votación en línea.

ATVotaciones opera enteramente dentro del servidor de Plone, aceptando votos en texto claro de los electores, generando un recibo de voto en el propio servidor y realizando el conteo también en el servidor mismo. Aunque el sistema protege la comunicación entre el elector y el servidor mediante SSL/TLS [IETF2008], la seguridad de la elección depende enteramente del servidor. Cualquier entidad con acceso administrativo al servidor podría modificar el sistema para alterar o revelar a un tercero los votos capturados (junto con la identidad de su elector) y, de hecho, cambiar arbitrariamente el protocolo de la elección. En el caso general, dichas modificaciones podrían ser muy difíciles de detectar e incluso realizadas de modo que no dejaran rastro alguno una vez concluida la elección. El recibo ofrecido por ATVotaciones no permite realmente verificación por parte del usuario, ya que al ser generado únicamente del lado del servidor, dicho servidor puede entregar al elector un recibo espurio correspondiente a una versión modificada de su voto, sin que éste tenga forma de detectar tal substitución.

Lo anterior se vuelve más preocupante en cuanto a que, tal como ha sido instalado y usado el sistema en el IMATE a la fecha, cualquier persona con acceso físico al servidor que aloja ATVotaciones puede fácilmente adquirir privilegios administrativos sobre éste. Esto sin considerar posibles intrusiones remotas, así como intentos de alterar los resultados de la elección o violar la privacidad del votante por parte de las propias autoridades realizando la elección.

Algunas acciones del sistema ATVotaciones hacen uso de criptografía básica, incluyendo el cifrado de votos. Estas medidas de seguridad son trivialmente subvertidas por un atacante con acceso privilegiado al servidor durante la elección, y redundantes contra un atacante que nunca tenga acceso privilegiado a éste.

ATVotaciones presenta problemas adicionales, entre ellos falta de compatibilidad con las últimas versiones del manejador de contenidos Plone y problemas a nivel de implementación. Sin embargo, es su diseño de seguridad fundamental el que llevó a la decisión de reemplazar, en vez de actualizar, dicho sistema.

1.3.2. Helios Voting

En el otro extremo del espectro, Helios Voting [Adida2008] es un sistema para la realización de elecciones en línea ampliamente reconocido como seguro, de acuerdo con el estado del arte. Este sistema ha sido usado también para llevar a cabo elecciones internas en el IMATE, pero sufre de falta de integración con el resto de la infraestructura de sistemas usada por el instituto, dificultando su uso.

Helios Voting hace uso de cifrado homomórfico para el conteo de votos, lo cual lo distingue de [Benaloh2006] y PloneVote, que basan su conteo en redes de mezcla. En términos generales, un conteo homomórfico de votos es un proceso mediante el cual los votos cifrados capturados en el sistema son combinados, sin ser descifrados, para producir un conteo de los resultados de la elección, cifrado bajo el mismo esquema y llave[3] que los votos. Este conteo cifrado es posteriormente descifrado para obtener los resultados de la elección en claro, sin jamás descifrar los votos en sí.

Una desventaja del uso de cifrado homomórfico, es que relaciona fuertemente el esquema utilizado para representar y contar los votos con el mecanismo de seguridad de la elección. Un conteo homomórfico correcto depende del esquema de representación y conteo de votos utilizado en la elección, y es un problema no trivial construir tal conteo dado un cierto esquema de representación y conteo de votos en texto claro[4].

El reglamento actual del IMATE contempla diversos esquemas de representación y conteo de votos, utilizados para distintos procesos de elección internos, algunos de los cuales pueden ser solo aproximados utilizando el esquema de votación presentado por Helios. En general, es razonable suponer que esquemas de votación diversos serán usados por distintas instituciones para diversos tipos de elecciones intra-institucionales. Por lo tanto, es deseable contar con un sistema en que la representación y conteo de votos sea independiente de los mecanismos utilizados para garantizar las propiedades de seguridad de la elección. El modelo de redes de mezcla, utilizado por PloneVote, tiene esta característica.

Nuestro interés por contar con una integración profunda con el manejador de contenidos Plone y con un soporte extensible para esquemas arbitrarios de representación y conteo de votos, nos llevó a basar el desarrollo de PloneVote en [Benaloh2006], en lugar de adaptar Helios.

1.4. Estructura de este trabajo

Los primeros tres capítulos que siguen a esta introducción describirán progresivamente los mecanismos criptográficos utilizados por PloneVote, junto con los detalles generales de su implementación en PloneVoteCryptoLib. Cada uno de estos capítulos será dividido en tres secciones, siguiendo un mismo esquema. La primera sección presentará, de forma intuitiva, las técnicas o mecanismos criptográficos expuestos en el capítulo. La segunda sección dará una construcción, a nivel matemático, de dichas técnicas o mecanismos, incluyendo los algoritmos generales que describen las operaciones que los

[3] Definiremos los conceptos de esquema criptográfico y llave en el Capítulo 2.

[4] Resultados recientes en criptografía fundamental sugieren que es posible realizar operaciones arbitrarias sobre datos cifrados de forma homomórfica [Gentry2009]. Sin embargo, implementaciones actuales de este principio son sumamente complejas y computacionalmente ineficientes.

conforman. Finalmente, la tercera sección presentará los detalles de nuestra implementación de estas técnicas o mecanismos dentro de PloneVoteCryptoLib, describiendo las clases y métodos que conforman tal implementación, junto con su interfaz y uso esperado.

El capítulo 2 introduce los conceptos y técnicas generales de cifrado de datos, haciendo particular énfasis en cifrado de llave pública (2.1). El capítulo explora a detalle el esquema de cifrado ElGamal (2.2), utilizado por PloneVote para cifrar los votos y como base para sus operaciones criptográficas más complejas. Presentaremos las clases de PloneVoteCryptoLib relacionadas con la implementación del esquema ElGamal básico (2.3). Adicionalmente, expondremos algunas características generales de ingeniería de nuestra biblioteca, las cuales tienen repercusiones importantes para nuestra implementación completa (2.3.2.1 y 2.3.2.3).

El capítulo 3 explora la teoría y práctica de cifrado de umbral en ElGamal. Describiremos el uso general del cifrado de umbral como un mecanismo para cifrar mensajes de modo que requieran de la cooperación de varias entidades para ser descifrados (3.1). Daremos la construcción del esquema de cifrado de umbral sobre ElGamal usado por PloneVote (3.2), aproximándolo gradualmente por esquemas más sencillos: describiremos primero el esquema de compartición de secreto de Shamir (3.2.1), seguido por un esquema que soporta descifrado distribuido pero genera sus llaves de forma centralizada (3.2.2.1) y, finalmente, el esquema de Pedersen para la generación de llaves de forma distribuida (3.2.2.2). Este capítulo introduce, además, el concepto de prueba en conocimiento cero (3.2.3.1) y su aplicación para la realización de un proceso de descifrado verificable (3.2.3.2 a 3.2.3.4). Presentamos también nuestra implementación de cifrado de umbral distribuido y verificable, incluida como parte de PloneVoteCryptoLib (3.3).

El capítulo 4 introduce el concepto de red de mezcla y el papel que ésta juega en la anonimización de una colección de votos (4.1). Describiremos qué significa que un proceso de mezcla sea verificable y por qué es tal una propiedad importante para nuestros propósitos (4.1.3). Daremos la construcción de una red de mezcla verificable por recifrado sobre el esquema ElGamal (4.2). Concluiremos, como de costumbre, con los detalles de nuestra implementación de redes de mezcla en PloneVoteCryptoLib (4.3).

El capítulo 5 sigue una estructura distinta a la de los tres anteriores. Dedicaremos la sección 5.1 a presentar el esquema de Elecciones Simples Verificables, tal como está dado originalmente en [Benaloh2006]. En la sección 5.2, expondremos el diseño de PloneVote de acuerdo con nuestros planes de implementación actuales. Nuestra descripción incluirá: los roles de los usuarios que participan en el sistema (5.2.1), los programas que lo constituyen (5.2.2), las fases de una elección sobre PloneVote (5.2.3) y las propiedades de seguridad provistas por dicho sistema (5.2.4). Con tal diseño en mente, en la sección 5.3 retomaremos PloneVoteCryptoLib, esta vez analizando su función como parte del sistema PloneVote completo.

Cerraremos el presente trabajo con unas breves conclusiones (6).

Capítulo 2

Cifrado de llave pública: ElGamal

2.1. Esquemas de cifrado de llave pública

2.1.1. Definiciones generales

El cifrado de datos es una técnica criptográfica fundamental, mediante la cual información arbitraria puede ser almacenada o transmitida en medios no físicamente seguros, manteniendo su confidencialidad. La información a ser transmitida o almacenada es primero transformada mediante algún proceso que hace imposible, o al menos impráctico, que algún adversario pueda recuperar su significado original. A dicho proceso se le denomina método de cifrado o simplemente cifrado, y se dice que los datos resultantes han sido cifrados. Aquellos usuarios legítimamente autorizados a leer los datos cifrados poseen una pieza de información adicional, llamada llave o clave de descifrado, que les permite recuperar el significado original de los datos cifrados. Al proceso de recuperar el mensaje original "en claro", a partir de los datos cifrados, se le llama descifrado.

Existen en la literatura varias formalizaciones distintas, mas esencialmente equivalentes, para describir un sistema de cifrado de datos (e.g. [Shannon1949], [MOV1996, Ch. 1] y [Reyzin2004, L. 1]). A continuación damos una formalización similar[1] a la dada en [Reyzin2004, L. 1], la cual usaremos durante el resto de este capítulo y extenderemos en capítulos posteriores para expresar las operaciones ahí descritas.

Definición 1. Esquema criptográfico: Un esquema criptográfico es una tupla $Ciph = (M, C, K_E, K_D, Enc, Dec)$ donde M, C, K_E y K_D son conjuntos finitos. Enc es un algoritmo tal que para cada entrada (m, k_e) con

[1] La principal diferencia está en que nuestra formalización hace énfasis adicional en las llaves de cifrado y descifrado.

$m \in M$ y $k_e \in K_E$ produce alguna salida $c \in C$. *Dec* es un algoritmo tal que para cada entrada (c', k_d) con $c' \in C$ y $k_d \in K_D$ produce alguna salida $m' \in M$.
Usamos la siguiente notación para los elementos del esquema criptográfico:

- M es el conjunto de posibles mensajes en claro, con cada $m \in M$ un mensaje.
- C es el conjunto de posibles textos cifrados, con cada $c \in C$ un texto cifrado.
- K_E es el espacio de llaves de cifrado, con cada $k_e \in K_E$ una llave de cifrado.
- K_D es el espacio de llaves de descifrado, con cada $k_d \in K_D$ una llave de descifrado.
- *Enc* es el algoritmo de cifrado y *Dec* es el algoritmo de descifrado.

Notemos que *Enc* y *Dec* son algoritmos, no funciones en el sentido estricto. En particular, existen numerosos esquemas interesantes[2] donde *Enc* es un algoritmo aleatorizado, capaz de producir diferentes salidas $c_1, ..., c_l$ en diferentes ejecuciones, dado el mismo par (m, k_e) como entrada.

Una noción importante para un esquema criptográfico (al menos uno de llave pública, ver 2.1.1.1) es la de par de llaves (o par de claves), formalizada a continuación.

Definición 2. Par de llaves: Un par de llaves para un esquema criptográfico $Ciph = (M, C, K_E, K_D, Enc, Dec)$ es una tupla (k_e, k_d) con $k_e \in K_E$ y $k_d \in K_D$, tal que, $\forall m \in M$, $Dec\left(Enc\left(m, k_e\right), k_d\right) = m$.

Dado lo anterior, podemos proponer una definición natural de correctitud de un esquema criptográfico.

Definición 3. Correctitud de un esquema criptográfico: Decimos que un esquema criptográfico $Ciph = (M, C, K_E, K_D, Enc, Dec)$ es correcto si y solo si $\forall k_e \in K_E \; \exists k_d \in K_D$ tal que (k_e, k_d) es un par de llaves.

Combinando las dos definiciones anteriores, tenemos que un esquema criptográfico es correcto cuando, para toda llave de cifrado válida en el esquema, existe una llave de descifrado correspondiente, la cual permite descifrar cualquier texto cifrado con esa llave de cifrado[3].

A partir de ahora, cuando hablemos de esquemas criptográficos consideraremos únicamente esquemas criptográficos correctos.

[2] Incluido ElGamal, nuestro principal ejemplo de interés para este trabajo.
[3] Notemos que, entre otras cosas, esto nos garantiza que el proceso de cifrado oculta, pero nunca destruye, información.

2.1.1.1. Esquemas de llave privada y esquemas de llave pública

Los esquemas criptográficos para el almacenamiento y transmisión de información confidencial pueden, a grandes rasgos, clasificarse en dos categorías: esquemas de llave privada (también llamados de cifrado simétrico) y esquemas de llave pública (cifrado asimétrico).

Definición 4. Esquema criptográfico de llave privada: Un esquema criptográfico correcto $Ciph = (M, C, K_E, K_D, Enc, Dec)$ es de llave privada si $K_E = K_D$ y, además, toda $k \in K_E = K_D$ es tal que (k, k) es un par de llaves. Esto es, si cada llave de cifrado es su propia llave de descifrado, y viceversa.

Es decir, en un esquema de llave privada existe una misma llave que es usada tanto por el algoritmo de cifrado como el de descifrado. Los esquemas de llave privada tienden a ser útiles en sistemas usados para almacenar información de forma segura. Dichos sistemas deben proteger la llave asociada a la información almacenada, ya sea guardándola físicamente en otro lugar o encadenándola con una contraseña que actúa como llave de la propia llave [TrueCrypt, dm-crypt, BitLocker]. Un esquema de llave privada también puede ser usado para comunicación, cuando existe algún canal seguro previo mediante el cual dos o más participantes pueden quedar de acuerdo en una misma llave. Un caso de particular interés, que consideraremos con más cuidado en 2.1.1.2, es el uso de un esquema de llave pública como un canal seguro para el intercambio de llaves de un segundo esquema de llave privada, usado finalmente para la transmisión de información.

Existen numerosos esquemas criptográficos de llave privada, tanto propuestos, como en uso actual. Algunos de los más notables son: DES [NIST1999], AES/Rijndael [DR1998, NIST2001], Blowfish [Schneier1994], Twofish [S et. al. 1998] y Serpent [ABK1998].

Definición 5. Esquema criptográfico de llave pública: Un esquema criptográfico correcto $Ciph = (M, C, K_E, K_D, Enc, Dec)$ es de llave pública si para cada uno de sus pares de llaves (k_e, k_d), tenemos $k_e \neq k_d$.

Es decir, un esquema criptográfico es de llave pública (asimétrico), si la llave usada para descifrar un mensaje es distinta de la usada para cifrarlo. Para esquemas de llave pública de uso práctico, buscamos además que obtener algún k_d que forme un par de llaves con un k_e dado sea imposible en la práctica. Esto nos permite dar a conocer k_e públicamente, incluyendo la posibilidad de que un atacante obtenga esta llave, sin que esto permita que el atacante descifre mensajes cifrados con k_e, siempre y cuando mantengamos la k_d correspondiente secreta. En tal escenario, llamamos a k_e la llave pública del par y a k_d la llave privada[4].

[4] Esto es solo el caso en escenarios en los cuales el esquema y el par de llaves son usados para la transmisión de mensajes. Muchos esquemas de llave pública pueden transformarse

Consideremos un sistema donde: (k_e, k_d) es un par de llaves de un esquema seguro (ver 2.1.1.3) de llave pública, k_e es conocida públicamente y k_d es conocida solo por una entidad A. Es sencillo ver que tal sistema permite a cualquier entidad B mandar mensajes secretos a A, sin temor de que otra entidad pueda leer dichos mensajes (veremos los detalles de ese escenario en 2.1.2).

Al igual que en el caso de los esquemas de llave privada, existen diversos esquemas de llave pública en uso, por ejemplo: RSA [RSA1978], ElGamal [ElGamal1985] y Paillier [Paillier1999].

2.1.1.2. Extendiendo el espacio de mensajes

Un detalle a notar de la definición 1, es que pedimos que todos los conjuntos sean finitos, incluidos M y C. Además, en la mayoría de los esquemas criptográficos, tomados en su forma más básica dada por dicha definición, el conjunto M que puede ser cifrado por Enc depende del conjunto de llaves K_E. Equivalentemente, el tamaño del mensaje que puede ser cifrado depende del tamaño de la clave de cifrado. Por otro lado, en la práctica, usualmente nos gustaría que las llaves fueran pequeñas comparadas con el tamaño de los mensajes que necesitamos cifrar.

Una forma de resolver el problema anterior es cifrar un mensaje largo por bloques. Es decir, dado un texto en claro μ, éste se codifica como una secuencia de mensajes $m_1, ..., m_k \in M$ y cada uno es cifrado independientemente con la misma llave k_e. Para recuperar μ, se descifra cada m_i con k_d de forma independiente también, y se decodifica μ a partir de $m_1, ..., m_k$. A esto se le conoce como cifrado por bloques y a cada m_i se le denota un "bloque" del mensaje original. En esquemas de cifrado de llave privada, el cifrado por bloques es la técnica más común para cifrar mensajes de longitud arbitraria y es un método ampliamente utilizado. Es importante notar que, como veremos en 2.1.1.3, el cifrado por bloques tiene importantes consecuencias para la seguridad de esquemas criptográficos.

En los esquemas de llave pública, una alternativa al cifrado por bloques es el cifrado híbrido. En un esquema de cifrado híbrido, se eligen un esquema de cifrado de llave pública y uno de llave privada. El esquema de llave privada es utilizado para cifrar, por bloques, cada mensaje largo μ que se quiera transmitir, con una nueva llave k. Posteriormente, k es cifrada usando el esquema de llave pública y adjuntada en tal forma a la versión cifrada de μ. Para descifrar el mensaje completo, el destinatario usa su llave k_d en el esquema de llave pública para obtener la llave k del mensaje, y posteriormente usa k para recuperar μ. Cuando un esquema de llave pública es usado para transmitir mensajes, sin consideraciones adicionales, generalmente se utiliza como parte de un esquema híbrido junto con un algoritmo de llave

para producir esquemas de firmas digitales, en los cuales los roles de las llaves se invierten (k_d es pública, mientras que k_e es privada). Este uso de los esquemas de llave pública queda fuera del alcance de este trabajo.

privada que se considera seguro. El cifrado híbrido se usa en la práctica porque los esquemas de llave privada tienden a permitir implementaciones más eficientes en computadoras actuales de Enc y Dec que los de llave pública, para un tamaño de llave seguro. Un ejemplo de un sistema que usa cifrado híbrido para transmitir mensajes secretos es GnuPG [FSF1999].

2.1.1.3. Seguridad en esquemas criptográficos

Intuitivamente, un esquema criptográfico es seguro en la medida de que un atacante, teniendo acceso a $c = Enc(m, k_e)$ y sin tener acceso a k_d, sea incapaz de recuperar m. Usualmente, también es razonable suponer que cualquier atacante posible tiene acceso a los conjuntos y algoritmos que definen el esquema criptográfico general. Es decir, las únicas piezas potencialmente secretas son el mensaje y las llaves usadas.

Algunos esquemas criptográficos requieren para su seguridad mantener los algoritmos Enc y Dec secretos. Sin embargo, dicho requerimiento se considera una debilidad importante de tales esquemas, pues cambiar los algoritmos es en general más dificil que cambiar las llaves. La noción de que un esquema criptográfico seguro debe conservar su seguridad aun cuando sus algoritmos (mas no las llaves) son conocidos por un potencial atacante fue enunciada por vez primera por Auguste Kerckhoffs en [Kerckhoffs1883] y se conoce como el Principio de Kerckhoffs. Una discusión más actual del tema puede encontrarse en [Schneier2002].

Es importante notar que la definición 3 de correctitud de un esquema criptográfico no nos garantiza absolutamente nada respecto a la seguridad del mismo. El siguiente ejemplo muestra un esquema criptográfico trivial que es correcto, de acuerdo a nuestra definición, pero claramente inseguro siguiendo la intuición anterior.

Ejemplo 6. Consideremos el esquema criptográfico

$$Ciph = (M, C, K_E, K_D, Enc, Dec)$$

con $M = C$, K_E y K_D arbitrarios y $Enc(m, k_e) = m$, $Dec(c, k_d) = c$. Dicho esquema es correcto, pues para cada $k_e \in K_E$, cualquier $k_d \in K_D$ es tal que $\forall m \in M$, $Dec(Enc(m, k_e), k_d) = m$.

Por otro lado, $c = Enc(m, k_e) = m$, por lo cual cualquier atacante con acceso a c conoce trivialmente m.

Una definición matemáticamente formal de seguridad en esquemas criptográficos es la de "Seguridad Perfecta" o "Seguridad de Shannon" ([Shannon1949], [Reyzin2004, L. 1]). A continuación damos la más sencilla de al menos dos formas conocidas y equivalentes de definir dicha propiedad.

Definición 7. Seguridad de Shannon: Un esquema criptográfico $Ciph = (M, C, K_E, K_D, Enc, Dec)$ satisface seguridad de Shannon si para cualesquiera dos mensajes $m_1, m_2 \in M$ y cualquier texto cifrado $c \in C$, tenemos:

$$\Pr_{k_e \in K_E}[Enc(m_1, k_e) = c] = \Pr_{k_e \in K_E}[Enc(m_2, k_e) = c]$$

Es decir, el esquema *Ciph* cumple con Seguridad de Shannon si cualesquiera dos mensajes en claro tienen la misma probabilidad de ser cifrados como el mismo texto cifrado, dada una elección al azar de la llave de cifrado.

En un esquema que cumple con Seguridad de Shannon, dado un único texto cifrado c, generado como $c = Enc(m, k_e)$ para alguna $k_e \in K_E$ sobre la cual no se tiene ninguna información adicional, un atacante, sin importar cual sea su poder de cómputo, no puede obtener información alguna sobre m a partir de c, ya que cualquier $m \in M$ válido tiene la misma probabilidad de ser transformado en c por Enc cuando k_e es tomada de K_E con una distribución de probabilidad uniforme.

Existen esquemas que cumplen con seguridad de Shannon, notablemente ciertos esquemas de *"One-time pad"* o *"Libreta de un solo uso"*[Shannon1949]. Sin embargo, la seguridad de Shannon tiene importantes limitaciones en la práctica.

Primero, la definición 7 no nos da ninguna información sobre qué ocurre cuando un atacante tiene acceso a más de un mensaje cifrado con la misma llave de cifrado k_e. De hecho, conocer el texto cifrado generado con una misma clave para múltiples mensajes relaciona las probabilidades de dichos mensajes, lo que dependiendo del esquema podría permitir obtener estos mensajes en claro o incluso la llave k_e, mediante criptoanálisis. Esto requiere transmitir de forma segura una nueva llave por cada mensaje transmitido en un esquema de llave privada, lo cual hace el manejo de llaves prohibitivamente complejo para la mayoría de los escenarios (más aún cuando consideramos que, como vimos en 2.1.1.2, un "mensaje" largo es en realidad probablemente una serie de varios mensajes $m_i \in M$).

Segundo, para que la Seguridad de Shannon garantice seguridad verdadera, es necesario que el atacante no posea ninguna información sobre k_e más allá de que éste sea cualquier elemento de K_E con probabilidad uniforme. En esquemas de llave pública, es razonable suponer que el atacante conoce k_e, lo cual hace imposible obtener Seguridad de Shannon en tales esquemas.

El principal problema con la propiedad de Seguridad de Shannon como una definición práctica de seguridad (al menos en esquemas de llave pública), es que es independiente del poder de cómputo del atacante. Para cualquier esquema criptográfico de llave pública donde el atacante conoce k_e, uno podría imaginar que éste cifra cada posible mensaje $m \in M$ con k_e cuantas veces sea necesario para obtener todos los posibles textos cifrados $Enc(m, k_e)$, y se detiene solo al encontrar aquel m que se cifra a un c dado. Es fácil ver que un esquema criptográfico no puede generar un mismo c para dos mensajes distintos con una misma llave de cifrado y aun así ser correcto (Dec tendría que "adivinar" de forma infalible de cuál mensaje se trata, sin ninguna información para distinguirlos). De tal modo, cualquier mensaje cifrado en un sistema de llave pública es descifrable por un atacante

con poder de cómputo no acotado que conoce la llave pública con la que se cifró el mensaje.

Una propiedad de seguridad más útil para evaluar este tipo de esquemas, es la de Seguridad Semántica o indistinguibilidad bajo texto claro elegido (*"indistinguishability under chosen-plaintext attack"* ó IND-CPA). Dicha propiedad fue formulada originalmente por Goldwasser y Micali, y puede ser pensada como un equivalente de Seguridad de Shannon cuando las acciones del adversario son polinomialmente acotadas en tiempo con respecto a su entrada [GM1984]. Damos a continuación una definición comúnmente usada de IND-CPA, similar a la dada en [Waters2009, L. 9].

Definición 8. Seguridad IDA-CPA: Suponemos un atacante A y un retador C con acceso a la información (conjuntos y algoritmos) de un esquema criptográfico correcto de llave pública $Ciph = (M, C, K_E, K_D, Enc, Dec)$. A y C participan en el siguiente juego:

1. C genera un par de llaves (k_e, k_d) en $Ciph$ y pública k_e (en particular, se la da a A)

2. A elije dos mensajes m_1, m_2 de la misma longitud[5] y pasa ambos a C.

3. C elije $b \in \{0, 1\}$ al azar uniformemente, y cifra $c = Enc(m_b, k_e)$, entregando c a A.

4. A elije $b' \in \{0, 1\}$

Durante la realización del juego anterior, A no debe realizar más que un número de pasos polinomial en la longitud de k_e. Un esquema criptográfico de llave pública es seguro en IDA-CPA si la probabilidad $Pr[b = b']$ de que A pueda adivinar b con b' no es significativamente mayor que $\frac{1}{2}$.

Esto es, A no debe ser capaz de adivinar, en tiempo polinomial, cuál de los dos mensajes fue cifrado por C. Claramente, solo aquellos esquemas en que Enc sea un algoritmo aleatorizado pueden llegar a satisfacer IDA-CPA. Si Enc es una función, A podría simplemente cifrar cada uno de los mensajes él mismo y compararlos con c para determinar b.

Un hecho interesante es que la propiedad de seguridad IDA-CPA es equivalente a la necesaria para que A no pueda adivinar b con probabilidad significativamente mayor que $\frac{1}{2}$, cuando el juego se modifica para permitir que A entregue dos secuencias de mensajes (de la misma longitud) a C, en vez de tan solo dos mensajes individuales y C elija una de las dos secuencias completas para cifrar y entregar cifrada a A [Waters2009, L. 9]. Por lo tanto, un esquema que es seguro bajo IDA-CPA mantiene su seguridad cuando el esquema de cifrado es usado para transmitir múltiples mensajes.

[5] Cada uno puede estar compuesto de varios mensajes en M, como se describe en 2.1.1.2.

En general, probar que un esquema de cifrado de llave pública satisface IDA-CPA de forma incondicional no es factible con herramientas matemáticas actuales[6]. Por otro lado, es posible mostrar que ciertos esquemas de cifrado son seguros en IDA-CPA si y solo si no existe un algoritmo eficiente (es decir, polinomial) para resolver un cierto problema matemático. Por ejemplo ElGamal, descrito en 2.2 y usado por PloneVoteCriptoLib, es seguro en IDA-CPA provisto que no exista un algoritmo eficiente para obtener el logaritmo discreto en un grupo cíclico. A pesar de numerosos intentos de matemáticos y criptoanalistas, tal algoritmo no ha sido encontrado a la fecha, lo que nos da un buen nivel de confianza en la seguridad de ElGamal, aun si no podemos tener ninguna certeza matemática al respecto.

2.1.2. Transmisión de mensajes en un esquema de llave pública

Dada la teoría anterior, buscamos a continuación mostrar el uso y consideraciones prácticas de un esquema de cifrado de llave pública para la transmisión de mensajes entre dos entidades.

Nuestro escenario es el siguiente: Supongamos que tenemos dos usuarios, Alicia y Beto[7]. Ambos usuarios cuentan con un dispositivo de cómputo personal en el cual confían, es decir, saben que dicho equipo ejecutará los programas que ellos le instruyan correctamente y no revelará o alterará información contenida en el mismo sin que su respectivo usuario lo sepa. Dichos equipos de cómputo se encuentran conectados uno al otro a través de una conexión no físicamente segura (internet, por ejemplo), y Alicia y Beto no tienen forma alguna de comunicarse mediante ningún otro canal. Suponemos, además, que existen uno o más atacantes potenciales en la red que pueden ver cualquier comunicación entre Alicia y Beto a través de la conexión antes mencionada. Por el momento consideraremos solo el caso de atacantes pasivos que pueden interceptar, mas no alterar, la comunicación entre Alicia y Beto. Beto desea mandar un mensaje secreto a Alicia, pero no quiere que ninguno de los atacantes pueda leer ese mensaje.

Si Alicia y Beto conocen un esquema criptográfico de llave pública seguro, entonces pueden tomar las acciones siguientes para comunicarse:

---

[6] Consideremos la famosa pregunta abierta en computación ¿$P = NP$?. Dado un algoritmo aleatorio Enc con tiempo de ejecución polinomial en k_e, trivialmente existe una forma en NP de saber si c es una posible salida para $Enc(m_i, k_e)$, explorando simultáneamente todas las posibles elecciones de Enc. Por lo tanto, si P fuese igual a NP, entonces existiría un algoritmo polinomial para saber si c es un posible cifrado de m_i, lo cual permite ganar el juego de IDA-CPA.

[7] Los clásicos "Alice" y "Bob" usados en la literatura en inglés para representar entidades A y B.

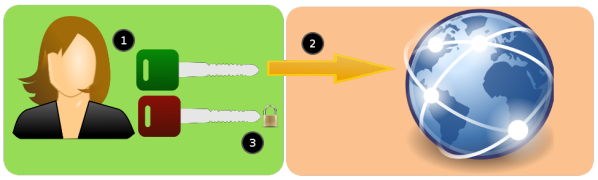

Figura 2.1.1: Alicia genera un par de llaves, publicando su llave pública en internet y almacenando su llave privada de forma segura.

Figura 2.1.2: Beto cifra su mensaje secreto con la llave pública de Alicia y envía el mensaje cifrado a través de un canal inseguro (e.g. Internet).

2.1.2.1. Establecimiento de llaves

Antes de que Beto pueda mandar un mensaje a Alicia de forma segura, ella debe establecer un par de llaves y hacer su llave pública accesible a Beto. Esta primera fase del escenario se muestra en la figura 2.1.1 y se describe a continuación.

Lo primero que debe ocurrir es que Alicia genere un par de llaves en el esquema criptográfico usado (1), con un tamaño de llave adecuado para obtener el nivel de seguridad deseado. Alicia entonces envía la llave pública a través de su conexión con Beto, en efecto publicándola a Beto y a cualquier atacante que se encuentre observando la conexión. Alternativamente, Alicia puede simplemente colocar su llave pública en un sitio de Internet de donde Beto (y cualquier atacante) puede recuperarla (2). La llave privada, en cambio, debe ser almacenada por Alicia de forma segura en su equipo personal, posiblemente encadenándola con una contraseña para seguridad adicional (3). La llave privada nunca debe ser transmitida por una conexión no segura.

2.1.2.2. Envío del mensaje

Para enviar un mensaje secreto a Alicia, Beto sigue los pasos de la figura 2.1.2. Primero, Beto construye el mensaje secreto "en claro" que quisiera enviar a Alicia de forma segura (1). Dicho mensaje no debe salir del equipo de cómputo confiable de Beto en tal estado, pues podría ser fácilmente leído por los atacantes observando los datos que Beto manda a través de su conexión con Alicia. En cambio, Beto obtiene la llave pública de Alicia y, mediante el

Figura 2.1.3: Un atacante que recupera el mensaje cifrado no puede leer su contenido, independientemente de si posee la llave pública o no. Por otro lado, Alicia, usando la llave privada, puede fácilmente recuperar la información del mensaje original a partir de los datos cifrados.

esquema de cifrado de llave pública conocido por ambos, la usa para cifrar el mensaje (2). Si el mensaje es largo, Beto puede simplemente cifrarlo por bloques (ver 2.1.1.2). Finalmente, el mensaje cifrado es transmitido a través de la conexión no segura hasta ser obtenido por Alicia (3).

2.1.2.3. Descifrado del mensaje

Como se observa en la figura 2.1.3, un atacante no debería ser capaz de descifrar el contenido del mensaje cifrado que Beto envió a Alicia (1). Es decir, no debe haber forma de que, dado dicho mensaje cifrado, el atacante pueda saber o aproximar cuál pudo ser el mensaje secreto original que Beto cifró. Esto debe ser cierto incluso si el atacante tiene acceso a la llave pública de Alicia (2), pues es esperable que dicho atacante la haya interceptado cuando Alicia la publicó para Beto. Solo Alicia, usando su llave privada que nunca fue transmitida por ningún canal inseguro, puede recuperar el contenido del mensaje mediante el algoritmo de descifrado (3).

Observemos que el hecho de que la llave pública no pueda ser usada para descifrar mensajes cifrados con esa misma llave, implica que Beto tampoco es capaz de descifrar ningún mensaje cifrado para Alicia (puede leer su propio mensaje solo porque tiene acceso a la copia "en claro" del mismo). Esto quiere decir que el mismo par de claves puede ser usado por Alicia para que múltiples otros usuarios se comuniquen con ella de forma secreta, sin que ninguno de estos usuarios pueda leer los mensajes enviados a Alicia por otro. Por otro lado, si ahora Alicia quisiera mandar mensajes a Beto, él tendría que generar su propio par de llaves y ambos tendrían que seguir todos los pasos anteriores, pero ahora con los roles invertidos.

En general, para que n usuarios intercambien mensajes secretos usando un esquema de llave pública, cada usuario requiere exactamente un par de llaves, independientemente de cuantos usuarios participen en el intercambio. Adicionalmente, una vez generado y transmitido un par de llaves, en la mayoría de los esquemas de llave pública éste puede ser usado para transmitir un número arbitrario de mensajes al usuario que posee la llave privada del

par[8].

2.1.2.4. Consideraciones de seguridad

Al describir el escenario anterior, omitimos mencionar algunos vectores de ataque que podrían permitir a un atacante leer el contenido de algún mensaje secreto enviado por Beto a Alicia, aun si el esquema criptográfico usado es seguro. Las dos principales consideraciones de seguridad que omitimos son: autenticidad de la llave pública y seguridad de la plataforma.

En nuestra descripción, supusimos un atacante pasivo, el cual simplemente observa los datos transmitidos a través del canal inseguro que usan Alicia y Beto para comunicarse. El escenario anterior es seguro contra tal atacante. Sin embargo, si permitimos también que el atacante modifique los mensajes que pasan por el canal, el atacante podría comprometer la seguridad de la comunicación de la siguiente forma:

Supongamos que el atacante intercepta la llave pública de Alicia cuando ella la publica originalmente y la reemplaza, sin que Beto se de cuenta, por una nueva llave pública generada por dicho atacante, y para la cual éste posee la llave privada correspondiente. Cuando Beto desea mandar un mensaje a Alicia, lo cifra con la llave pública del atacante, pensando que dicha llave corresponde a Alicia. El atacante solo debe entonces interceptar el mensaje cifrado, descifrarlo con su llave privada, leerlo, cifrarlo de nuevo con la llave pública real de Alicia y, finalmente, retransmitirlo a Alicia. Si el atacante puede transparentemente alterar cualquier mensaje entre Alicia y Beto, y está siempre presente en el canal de comunicación, entonces puede interceptar los mensajes entre ambos de esta forma, sin que Alicia y Beto se den cuenta.

Lo que permite el ataque anterior, y varios similares, es la falta de autentificación de la llave pública: Beto no tiene forma de saber que la llave pública que está obteniendo, por un canal inseguro, pertenece realmente a Alicia. Existen varias soluciones de uso común para autentificar llaves en un esquema de llave pública: entidades certificadoras centrales [IETF2008], redes de confianza [FSF1999] y autentificación basada en un secreto previo [BGB2004], por dar algunos ejemplos. PloneVote utiliza una estrategia ligeramente diferente a las antes mencionadas para garantizar la autenticidad de la llave pública de una elección (la cual no "pertenece" a ningún único usuario). Dicha estrategia se describe en 3.3.1. De momento, es tan solo importante que el lector esté consciente de esta vulnerabilidad en el escenario simple anteriormente descrito.

Un segundo supuesto en nuestro escenario, es que los equipos de cómputo que utilizan Alicia y Beto para generar sus llaves, mensajes y realizar los procesos de cifrado y descifrado, son confiables, y que el atacante no tiene

[8] La llave pública solo es invalidada (revocada), por razones de seguridad, cuando el usuario que generó el par de llaves tiene motivos para pensar que la llave privada correspondiente pudo haber caído en manos de un atacante.

acceso al estado interno de dichos equipos o a ninguna información que no es explícitamente enviada por los usuarios a través del canal inseguro. Llamamos a este supuesto el supuesto de seguridad de la plataforma, en donde la seguridad del sistema completo requiere que los clientes de cómputo usados por los usuarios sean seguros y confiables. PloneVote depende de la seguridad de la plataforma para su seguridad. En 5.2.4.8, argumentamos que, para el tipo de escenarios para los cuales está pensado PloneVote, es razonable suponer la seguridad de la plataforma, al menos del lado del cliente[9].

2.2. Teoría: Esquema ElGamal

2.2.1. Bases y uso de ElGamal

El esquema de cifrado de llave pública ElGamal fue propuesto en 1984 por Taher ElGamal [ElGamal1985], y es uno de los esquemas de llave pública más frecuentemente utilizados en la práctica. En particular, ElGamal es usado en GnuPG [FSF1999] y OpenPGP [IETF1998]. En parte debido a lo anterior, la seguridad de ElGamal ha sido ampliamente evaluada, y los problemas en que ésta se basa se encuentran entre los más cuidadosamente analizados por la comunidad criptográfica. A pesar de dicho escrutinio, no se conocen, a la fecha, ataques efectivos contra implementaciones adecuadas de ElGamal, lo cual aumenta nuestra confianza en la seguridad del esquema. Adicionalmente, provisto que el problema de Diffie-Hellman (ver 2.2.2) no tenga solución polinomial, se sabe que ElGamal cumple seguridad IDA-CPA [Waters2009] (ver 2.1.1.3).

ElGamal posee, además, varias propiedades interesantes que lo hacen útil como componente de un protocolo seguro para elecciones en línea, las cuales exploraremos en capítulos siguientes. En particular, es la base de SVE [Benaloh2006] y, por tanto, de PloneVote. En esta sección describiremos a detalle los fundamentos (2.2.2), algoritmos (2.2.3 y 2.2.4) y consideraciones de seguridad (2.2.5) en relación a ElGamal como esquema criptográfico en abstracto. La sección 2.3, detalla la implementación de ElGamal básico en PloneVoteCriptoLib.

2.2.2. Los problemas del logaritmo discreto y de Diffie-Hellman

ElGamal basa su seguridad en la presunta dificultad de resolver el problema de Diffie-Hellman, el cual a su vez se basa en la dificultad de obtener

[9] Solo los equipos de cómputo personales de los votantes, miembros de la comisión y al menos un auditor honesto se suponen confiables. PloneVote no requiere confiar en la seguridad de servidores remotos.

el logaritmo discreto en el grupo cíclico $\mathbb{Z}_p^*$ [10]. La información dada a continuación sobre ambos problemas está tomada de [MOV1996, Ch. 3].

Definición 9. Dado cualquier entero $n \in \mathbb{Z}$, $\mathbb{Z}_n$ es el conjunto $\{0, ..., n-1\}$ de los enteros módulo n. El grupo multiplicativo de $\mathbb{Z}_n$ es $\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n | gcd\,(a, n) = 1\}$ con la operación de multiplicación módulo n. En particular, dado p primo, $\mathbb{Z}_p^* = \{1, ..., p-1\}$.

Es sencillo ver que $\mathbb{Z}_n^*$ es un grupo, observando que la multiplicación módulo n satisface cerradura, asociatividad, identidad e inverso en $\mathbb{Z}_n^*$. Adicionalmente, $\mathbb{Z}_p^*$, donde p es un número primo, es siempre un grupo cíclico. Es decir, existe al menos un generador $g \in \mathbb{Z}_p^*$, tal que cualquier $\alpha \in \mathbb{Z}_p^*$ puede ser escrito como $\alpha = g^i \ mod\ p$ para algún entero $i \in \{0, ..., p-2\}$ [Rotman2002, Teo. 3.30].

El orden de $\mathbb{Z}_n^*$ queda definido como el número $|\mathbb{Z}_n^*|$ de elementos en $\mathbb{Z}_n^*$. Por lo tanto, el orden de $\mathbb{Z}_p^*$ con p primo es $p-1$.

Existen algoritmos eficientes [MOV1996, Alg. 2.143] para obtener la exponenciación módulo n de cualquier número entero. En particular, para $\alpha \in \mathbb{Z}_p^*,\ i \in \mathbb{Z}$, podemos obtener $\alpha^i \ mod\ p$ en $O\,(log\ i)$. Es decir, la exponenciación módulo n es lineal (por tanto, polinomial) con respecto al tamaño de su entrada (la representación binaria de α e i). Por otro lado, no se conoce un algoritmo polinomial para el problema opuesto: obtener i a partir de α y α^i. A este segundo problema se le denomina el problema del logaritmo discreto.

Definición 10. Problema del logaritmo discreto: Dado un primo p, un generador g de $\mathbb{Z}_p^*$ y un elemento $\beta \in \mathbb{Z}_p^*$ cualquiera, obtener x, $0 \leq x \leq p-2$, tal que $g^x = \beta \ mod\ p$.

En 2.2.5, describiremos brevemente los mejores algoritmos conocidos para atacar este problema, dependiendo de la estructura del grupo cíclico $\mathbb{Z}_p^*$. Sin embargo, es importante mencionar desde ahora que en el caso general, no se conoce ningún algoritmo polinomial capaz de resolver el problema del logaritmo discreto.

Aunque la seguridad de ElGamal está íntimamente relacionada con la dificultad del problema del logaritmo discreto, el problema fundamental en que se basa ElGamal es uno ligeramente distinto. Este problema es conocido como el problema de Diffie-Hellman y enunciado continuación.

Definición 11. Problema de Diffie-Hellman: Dado un primo p, un generador g de $\mathbb{Z}_p^*$ y dos elementos $g^a \ mod\ p$ y $g^b \ mod\ p$ en $\mathbb{Z}_p^*$, obtener $g^{ab} \ mod\ p$.

[10] Ambos problemas pueden ser generalizados a grupos cíclicos arbitrarios. Sin embargo, por simplicidad, nos restringiremos al caso de $\mathbb{Z}_p^*$, usado en ElGamal tal como fue descrito en [ElGamal1985] y como es usualmente implementado en la práctica, incluyendo en PloneVoteCriptoLib.

Es fácil ver que si pudiéramos resolver el problema del logaritmo discreto de forma eficiente, podríamos también dar una solución eficiente al problema de Diffie-Hellman: simplemente usamos nuestra solución del logaritmo discreto para obtener a y b a partir de $g^a \ mod \ p$ y $g^b \ mod \ p$, multiplicamos estos elementos para obtener ab, y finalmente calculamos $g^{ab} \ mod \ p$.

No se sabe si existe o no alguna solución eficiente al problema de Diffie-Hellman que no involucre resolver el problema del logaritmo discreto, así que no podemos realmente decir que la seguridad de ambos problemas es equivalente. Sin embargo, hasta la fecha, no se conoce ninguna solución más eficiente para el problema de Diffie-Hellman que las que se obtienen inmediatamente de los algoritmos para solucionar el problema del logaritmo discreto.

En particular, la seguridad de ElGamal se basa en que para $p = 2q+1$, con p y q primos suficientemente grandes[11], no existe solución eficiente conocida para el problema de Diffie-Hellman en $\mathbb{Z}_p^*$.

2.2.3. Generación de llaves en ElGamal

Consideraremos primero que una instancia del esquema ElGamal está dada por una pareja (p, g), donde $p = 2q + 1$, con p y q primos, y g un generador de $\mathbb{Z}_p^*$. Cada par de llaves (definición 2) es generado dentro de una instancia de ElGamal específica. Por otro lado, múltiples pares de llaves pueden ser generados para la misma instancia de ElGamal, lo que los hace compatibles entre sí para ciertas operaciones que veremos más adelante, como la generación de una llave pública de umbral[12].

Para facilitar nuestra discusión, llamaremos en adelante a $p = 2q+1$, con p y q primos, un primo seguro. La razón para ello se menciona en la sección 2.2.5.

Definición 12. Primo seguro: p es un primo seguro si $p = 2q + 1$, con p y q primos.

Damos un algoritmo general para construir una instancia del esquema ElGamal (Alg. 2.1). Este algoritmo supone que podemos determinar si un número es primo y dar un generador para el grupo $\mathbb{Z}_p^*$ cuando p es un primo seguro. Estas operaciones serán discutidas más adelante en las secciones 2.2.3.1 y 2.2.3.2. Notemos que el algoritmo toma como parámetro $nbits$ y regresa p primo seguro de tamaño $nbits$ (es decir, $2^{nbits-1} < p < 2^{nbits}$).

Dada una instancia particular de ElGamal, dicha instancia produce un esquema criptográfico de llave pública de la forma $Ciph = (M, C, K_E, K_D, Enc, Dec)$ (ver definición 1), donde $M = K_E = K_D = \{1, ..., p-1\}$, los elementos de $\mathbb{Z}_p^*$, y $C = \{1, ..., p-1\} \times \{1, ..., p-1\}$ [13]. El algoritmo 2.2 genera un par de

[11] Ver sección 2.2.5.

[12] Algunos autores (e.g. [MOV1996]) consideran p y g como parte de la llave pública. Sin embargo, debido a las operaciones que buscamos describir más adelante sobre llaves, nos conviene pensar que p y g son fijos dentro de una misma colección de llaves usadas para una misma elección.

[13] Enc y Dec serán definidos en 2.2.4

Algoritmo 2.1 InstanciaElGamal(nbits)

1. Generamos p:
   a) Tomamos un número q al azar en el intervalo $\left(2^{nbits-2}, 2^{nbits-1}\right)$
   b) Revisamos si q es primo (ver sección 2.2.3.1). Si no lo es, repetimos $(1a)$ hasta que lo sea
   c) Tomamos $p = 2q + 1$
   d) Revisamos si p es primo. Si no lo es, repetimos todo el paso 1
2. Tomamos g un generador de $\mathbb{Z}_p^*$ (alg. 2.3)
3. Devolvemos (p, g) como una instancia de ElGamal

Algoritmo 2.2 GenerarLlaves(p,g)

1. Tomamos un entero al azar $a \in [2, p-2]$
2. Obtenemos $b = g^a \ mod \ p$
3. Devolvemos $k_d = a$ como la llave privada y $k_e = b$ como la llave pública

llaves para dicho esquema criptográfico, donde la llave privada es un $a \in \mathbb{Z}_p^*$ cualquiera[14] y la correspondiente llave pública es $g^a \ mod \ p$. Por lo mencionado en 2.2.2, obtener la llave pública a partir de la llave privada es sencillo, mientras que el procedimiento inverso requiere resolver el problema del logaritmo discreto, el cual no tiene solución eficiente en la práctica cuando p es primo seguro y suficientemente grande.

2.2.3.1. Pruebas de primalidad

El algoritmo 2.1 requiere que podamos determinar, de forma eficiente, si un número cualquiera es un primo. Desafortunadamente, probar primalidad de un número arbitrario es un problema difícil, para el cual no se conoce ninguna solución en tiempo polinomial en el caso general.

Existen algoritmos para construir primos de forma verificable (e.g. el algoritmo de Mauer [MOV1996, Alg. 4.62]), los cuales funcionan restringiéndose a generar y probar números con formas particulares para los cuales podemos construir un certificado de primalidad a la vez que construimos el primo. Este certificado puede ser verificado posteriormente y de manera eficiente, junto con el número, para probar con toda certeza que se trata de

[14] En el algoritmo no permitimos que a sea 1 ó $p-1$, porque eso causaría que la llave pública fuera g ó 1, respectivamente, y sería trivial darse cuenta de cual es la llave privada en tales casos.

un primo. Un problema en generar primos verificables para ElGamal es, sin embargo, que no hay una forma sencilla de generar un primo verificable que además sea claramente un primo seguro. Es decir, aun si generamos p como un primo verificable, no tenemos forma sencilla de saber si $q = (p-1)/2$ es primo o no.

El algoritmo de Miller-Rabin [MOV1996, Alg. 4.24], en cambio, es un algoritmo probabilístico que, dado un número arbitrario n y una cierta probabilidad ρ, responde de la siguiente manera: si n es primo, Miller-Rabin siempre responderá que es primo; si n es compuesto, entonces Miller-Rabin responderá que n es compuesto con probabilidad al menos $1-\rho$, y que es primo con probabilidad a lo más ρ. Dado que Miller-Rabin es eficiente incluso para ρ relativamente pequeña, podemos usarlo, al generar nuestra instancia de ElGamal, para revisar que p y q son primos con muy alta probabilidad. En la práctica, $\rho = 2^{-256}$ reduce la probabilidad de obtener una instancia "rota" de ElGamal, a la equivalente a que un atacante adivine completamente al azar la llave de la mayoría de los esquemas criptográficos simétricos en uso actual. Es posible, en un tiempo razonable, ejecutar Miller-Rabin con tal ρ.

El algoritmo de Miller-Rabin se basa en el siguiente par de hechos:

Hecho 13. *Sea n un primo impar. Escribimos $n-1 = 2^k m$ para algún $k \geq 1$ y m impar. Para cualquier entero x tal que $gdc(x,n) = 1$, ocurrirá que $x^m = 1 \ (mod\ n)$ ó $x^{m2^i} = -1 \ (mod\ n)$, para algún $0 \leq i < k$.*

y

Hecho 14. *Sea $n > 9$ un entero impar positivo compuesto. Escribimos $n-1 = 2^k m$ para algún $k \geq 1$ y m impar. El número de elementos en $A = \{x | gcd(x,n) = 1\}$, tales que $x^m = 1 \ (mod\ n)$ ó $x^{m2^i} = -1 \ (mod\ n)$, para algún $0 \leq i < k$, es $\leq \frac{1}{4}$ de los elementos de A.*

Por tanto, dado n, si probamos números $x \in A$ al azar, encontrando cada vez que $x^m = 1 \ (mod\ n)$ ó $x^{m2^i} = -1 \ (mod\ n)$, para algún $0 \leq i < k$, rápidamente reduciremos la probabilidad de que n sea compuesto.

Una discusión mucho más completa de las bases para esta prueba de primalidad, incluyendo las pruebas de los "hechos" anteriores, puede ser encontrada en [CP2001]. Un listado detallado de los pasos del algoritmo de Miller-Rabin para revisar primalidad con probabilidad de error a lo más ρ, aparece en [MOV1996, Ch. 4].

2.2.3.2. Obtención del generador

Dado ya un primo seguro p, el algoritmo 2.1 nos pide además encontrar un generador g del grupo cíclico $\mathbb{Z}_p^*$.

Sea $a \in \mathbb{Z}_p^*$ un elemento arbitrario del grupo cíclico, buscamos determinar si a es o no un generador del grupo. Ahora, el orden del grupo $\mathbb{Z}_p^*$ es $p-1$, por lo que sabemos que $a^{p-1} = 1 \ mod\ p$. Por otro lado, a podría generar un

Algoritmo 2.3 EsGenerador(g,p)

1. Verificamos que g esté en $\mathbb{Z}_p^*$ (es decir, $1 \leq g \leq p-1$)
2. Obtenemos $q = (p-1)/2$
3. Si $g^2 = 1 \ mod \ p$, devolvemos "falso"
4. Si $g^q = 1 \ mod \ p$, devolvemos "falso"
5. En caso contrario, devolvemos "verdadero" (g es un generador de $\mathbb{Z}_p^*$)

Algoritmo 2.4 $Enc\,(p, g, k_e, m)$

1. Generamos un entero k al azar en $[1, p-2]$
2. Calculamos $\gamma = g^k \ mod \ p$ y $\delta = m \cdot (k_e)^k \ mod \ p$
3. Regresamos $c = (\gamma, \delta)$ como el texto cifrado

subgrupo propio dentro de $\mathbb{Z}_p^*$, en cuyo caso no sería un generador del grupo completo. En general, deberíamos verificar que no existe ningún $i < p-1$ para el cual $a^i = 1 \ mod \ p$ (lo que significaría que a genera un subgrupo de $\mathbb{Z}_p^*$ de orden $i < p-1$). Sin embargo, tenemos, por el teorema de Lagrange[15], que:

Teorema 15. ***Teorema de Lagrange:*** *Si G es un grupo finito, y H es un subgrupo de G, entonces el orden $|H|$ de H, divide al orden $|G|$ de G.*

Ahora, dado que p es un primo seguro, el orden de $\mathbb{Z}_p^*$ es $p-1 = 2q$, con q primo. Esto quiere decir que los únicos divisores del orden de $\mathbb{Z}_p^*$ son 2, q y $2q$. Basta entonces revisar si a genera un subgrupo de orden 2 u orden q. En caso contrario, a no genera ningún subgrupo propio de $\mathbb{Z}_p^*$, y debe, por tanto, ser un generador para el grupo $\mathbb{Z}_p^*$ completo. Basándonos en este hecho, es sencillo ver que el algoritmo 2.3 determina correctamente si un elemento es o no un generador de $\mathbb{Z}_p^*$.

2.2.4. Cifrado y descifrado en ElGamal

En la sección anterior vimos cómo generar una instancia de un esquema ElGamal, así como un par de llaves válido dentro de dicha instancia. Falta tan solo describir el proceso de cifrar y descifrar un mensaje $m \in M = \{1, ..., p-1\}$. El listado 2.4 da el algoritmo de cifrado *Enc*, mientras que el listado 2.5 da el algoritmo de descifrado *Dec*.

[15] Podemos encontrar una prueba de dicho teorema en, por ejemplo, [Herstein1975].

Algoritmo 2.5 $Dec\,(p, g, k_d, c)$

1. Extraemos (γ, δ) de c
2. Usando la llave privada k_d, calculamos $\gamma^{-k_d}\ mod\ p = \gamma^{p-1-k_d}\ mod\ p$
3. Recuperamos $m = \left(\gamma^{-k_d}\right) \cdot \delta\ mod\ p$

Veamos primero que el proceso de cifrado y descifrado es correcto. Dado un par de llaves generado por el algoritmo 2.2, tenemos que $(k_e, k_d) = (g^a\ mod\ p, a)$ para alguna $a \in \mathbb{Z}_p^*$. Por tanto, $Enc\,(p, g, k_e, m)$ genera $c = (\gamma, \delta) = \left(g^k\ mod\ p,\ m \cdot g^{ak}\ mod\ p\right)$. Ahora, al descifrar, obtenemos $\gamma^{-k_d} = \gamma^{-a} = g^{-ak}$ y lo multiplicamos por δ, obteniendo:

$$\left(\gamma^{-k_d}\right) \cdot \delta \equiv \left(g^{-ak}\right)\left(mg^{ak}\right) \equiv m\ \ (mod\ p)$$

Lo cual recupera el mensaje original.

Notemos que si pudiéramos resolver el problema de Diffie-Hellman (ver sección 2.2.2) y obtener $g^{ak}\ mod\ p$ a partir de $k_e = g^a\ mod\ p$ y $\gamma = g^k\ mod\ p$, podríamos descifrar cualquier c sin conocer $k_d = a$ [16].

Dados los mismos parámetros (p, g, k_e, m), el texto cifrado c regresado por Enc depende de la k aleatoria escogida. Es decir, Enc puede cifrar un mismo mensaje como múltiples textos cifrados distintos, usando la misma llave pública. Todos estos textos cifrados se descifran al mismo mensaje original mediante Dec, usando llave privada apropiada. Esto es, existe una relación uno a muchos entre los elementos de M y C para un esquema ElGamal. Una consecuencia de lo anterior es que c es más grande que m para cada mensaje cifrado (dos veces más grande, de hecho, pues cada c es una pareja de elementos en $\mathbb{Z}_p^*$). Esta relación de uno a muchos entre mensajes en claro y textos cifrados es parte de lo que permite que ElGamal sea IDA-CPA seguro bajo el supuesto de que no existe solución polinomial al problema de Diffie-Hellman (ver secciones 2.1.1.3, 2.2 y [Waters2009]), y es además una propiedad fundamental necesaria para implementar las redes de mezcla usadas por PloneVote (Capítulo 4).

2.2.5. Seguridad de ElGamal

Como ya mencionamos en 2.2.2, la seguridad de ElGamal se basa en la dificultad de resolver el problema de Diffie-Hellman, cuyas soluciones más eficientes conocidas a la fecha pasan por resolver el problema del logaritmo discreto. Una discusión detallada de dicho problema y los algoritmos conocidos para resolverlo (en el caso general y en ciertos casos específicos de

[16] Podemos obtener $g^{-ak}\ mod\ p$ de $g^{ak}\ mod\ p$ como sigue: $\left(g^{ak}\right)^{p-2} \equiv g^{ak(p-1)-1(ak)} \equiv \left(g^{ak}\right)^{p-1}\left(g^{-ak}\right) \equiv g^{-ak}\ \ (mod\ p)$

interés), puede encontrarse en [Studholme2002]. Varios de estos algoritmos también son mencionados en [MOV1996, Ch. 3]. A continuación, mencionaremos algunos de los métodos para tratar el problema del logaritmo discreto y sus implicaciones para la seguridad de ElGamal. Por razones de espacio omitiremos aquí los detalles de los algoritmos o las pruebas respecto a su tiempo de ejecución.

En el caso general de un grupo G de orden n, los mejores algoritmos conocidos para resolver el problema del logaritmo discreto (e.g. el algoritmo de paso largo y paso corto de Shanks) toman tiempo $O\left(\sqrt{n}\right)$, lo que es exponencial en el tamaño $|n| = \Theta\left(lg\left(n\right)\right)$ de la entrada. De hecho, ha sido demostrado que, si tenemos un grupo G arbitrario del cual no conocemos más estructura y para el cual podemos únicamente realizar la operación del grupo y el cálculo de inversos de elementos en el grupo, entonces las soluciones más rápidas posibles para el problema son $\Theta\left(\sqrt{n}\right)$.

Por otro lado, si tenemos más información sobre G, por ejemplo, que $G = \mathbb{Z}_p^*$ con p primo, existen algunos algoritmos sub-exponenciales para resolver el problema del logaritmo discreto. En particular, si lo único que sabemos es que p es primo, el método de cálculo de índices (*"index calculus"*), nos da un algoritmo sub-exponencial, pero super-polinomial para obtener el logaritmo discreto en $\mathbb{Z}_p^*$.

Decir que el método de cálculo de índices es sub-exponencial pero super-polinomial, quiere decir que el algoritmo resultante es $o\left(\left(c_1\right)^p\right)$ para cualquier $c_1 > 1$, pero $\omega\left(p^{c_2}\right)$ para cualquier c_2. En notación L (ver [Studholme2002]), el tiempo de ejecución del método de cálculo de índices puede ser caracterizado como $L_p\left(\frac{1}{2}, c\right)$.

El método de cálculo de índices consta de tres fases, las primeras dos de las cuales solo dependen de p y g, y no del elemento del cual se desea calcular el logaritmo discreto. Adicionalmente, la componente principal del tiempo de ejecución del cálculo de índices ocurre en esas primeras dos fases. Dado el precómputo dependiente solo de p y g, el algoritmo para obtener el logaritmo discreto de un nuevo elemento $\alpha \in \mathbb{Z}_p^*$ es sumamente eficiente. Esto tiene la implicación de que, para un atacante, ejecutar el método de cálculo de índices en una instancia particular de ElGamal (e.g. una particular elección de parámetros (p, g)), es suficiente para romper la seguridad de cualquier par de llaves o mensaje cifrado dentro de dicha instancia, sin gastar tiempo adicional significativo en atacar la seguridad de cada llave privada o mensaje cifrado.

Un tercer tipo de ataque que nos conviene considerar es el método de Pohlig-Hellman. Este método se basa en obtener una factorización $p - 1 = p_1^{e_1}, p_2^{e_2}, ..., p_k^{e_k}$, del orden $p - 1$ de $\mathbb{Z}_p^*$, con p, p_i primos. Dada dicha factorización, el método de Pohlig-Hellman reduce el problema del logaritmo discreto en $\mathbb{Z}_p^*$ a resolver el mismo problema para una serie de subgrupos de orden p_i para cada p_i con $i = \{1, ..., k\}$. El algoritmo completo es $O\left(\sum_{1 \leq i \leq k} e_i \left(lg\ n + \sqrt{p_i}\right)\right)$. Si $p - 1$ puede ser factorizado en factores sufi-

Bits de seguridad en un esquema de llave privada	Seguridad equivalente en un esquema de llave pública
80 bits	1024 bits
112 bits	2048 bits
128 bits	3072 bits
192 bits	7680 bits
256 bits	15360 bits

Cuadro 2.1: Tamaños de llave pública y su equivalencia contra esquemas de llave privada, según NIST

cientemente pequeños, el método de Pohlig-Hellman da una solución eficiente al problema del logaritmo discreto en $\mathbb{Z}_p^*$. Sin embargo, si al menos un p_i es suficientemente grande, $\sqrt{p_i}$ se vuelve el término dominante del tiempo de ejecución del algoritmo, negando su ventaja.

El método de Pohlig-Hellman es la razón por la cual usamos primos seguros (ver definición 12) en nuestra implementación de ElGamal. Dado tal primo, el orden de $\mathbb{Z}_p^*$ es $2q$, con q primo, y por tanto el tiempo de ejecución de Pohlig-Hellman es $\Omega\left(\sqrt{q}\right) \equiv \Omega\left(\sqrt{p}\right)$.

2.2.5.1. Parámetros seguros en la práctica

Suponiendo el uso de un primo seguro p, el ataque más eficiente conocido contra ElGamal en la práctica utiliza el cálculo de índices. Aunque tal método es super-polinomial, es significativamente más eficiente que búsqueda aleatoria de la llave privada $k_d \in \mathbb{Z}_p^*$, lo cual hace que la longitud de llaves necesaria para alcanzar seguridad en ElGamal sea considerablemente mayor que las longitudes comúnmente usadas en esquemas simétricos o de llave privada (e.g. 128 o 256 bits en AES [NIST2001]).

Adicionalmente, el método de cálculo de índices permite romper una instancia de ElGamal para una determinada p y g una sola vez y, dado esto, poder descifrar cualquier mensaje cifrado con cualquier llave pública en dicha instancia. Dado que varias implementaciones de ElGamal utilizan la misma instancia del esquema para un número significativo de sus operaciones o, incluso, mantienen estos parámetros entre distintas instalaciones, la ganancia de un atacante que pueda completar las primeras dos fases del método de cálculo de índices para una cierta instancia de ElGamal es alta. Es razonable entonces pedir, para que una instancia de ElGamal sea segura, que p sea lo bastante grande como para que no sea factible ejecutar el algoritmo de cálculo de índices para p y un generador g de $\mathbb{Z}_p^*$ durante el tiempo de vida total del sistema, más cualquier periodo de tiempo en que el descifrado no autorizado de algún mensaje cifrado dentro del sistema pueda considerarse perjudicial.

Las recomendaciones respecto al tamaño de llaves para ElGamal varían. GnuPG, por ejemplo, recomienda llaves de entre 1024 y 3072 bits, con 2048 siendo el valor sugerido por omisión [GPG2008]. Por su parte, NIST [NIST2007] publica las equivalencias de seguridad dadas en la tabla 2.1 y, por tanto, recomienda el uso de llaves de al menos 3072 bits en esquemas de llave pública de los cuales se requieren garantías fuertes de seguridad.

2.3. Implementación: Cifrado y descifrado ElGamal en PloneVoteCryptoLib

2.3.1. Descripción y uso

La biblioteca de herramientas criptográficas de python, Python Cryptography Toolkit ó pycrypto [pycrypto], es una de las bibliotecas criptográficas más comúnmente utilizadas para programas en python, e incluye implementaciones de numerosos algoritmos criptográficos y funciones elementales comúnmente utilizadas por éstos. PloneVoteCryptoLib utiliza pycrypto para generar números aleatorios, probar la primalidad de un entero dado y generar huellas digitales seguras mediante el algoritmo SHA256.

Sin embargo, PloneVoteCryptoLib provee su propia implementación del esquema ElGamal, en vez de utilizar la dada por pycrypto o cualquier otra biblioteca criptográfica de propósito general. Tomamos esta decisión, debido a que el resto de los algoritmos utilizados por el protocolo de votaciones de PloneVote, incluyendo el cifrado de umbral (Capítulo 3) y redes de mezcla (Capítulo 4), requieren de acceso a parámetros internos de los objetos del esquema ElGamal utilizado. La interfaz pública de la implementación de ElGamal dada en pycrypto no es suficientemente expresiva para tal propósito. De usar tal implementación, nos veríamos forzados a depender de los detalles de implementación internos de pycrypto, que están sujetos a cambios de una versión a otra de la biblioteca.

Dado lo anterior, fue nuestra elección el construir nuestra propia implementación de ElGamal en PloneVoteCryptoLib, con la interfaz que nosotros requeríamos. Por supuesto, en los casos en que podíamos depender de las funciones elementales públicas de pycrypto (como en el caso de generar números aleatorios o detectar la primalidad de p y q), aprovechamos dicha funcionalidad en lugar de re-implementarla.

En la sección 2.3.2 detallaremos las clases dentro de PloneVoteCryptoLib dedicadas a la implementación del esquema ElGamal básico. Para facilitar el acceso a estas funciones se incluyen dentro del paquete *plonevotecryptolib.tools* una serie de utilidades ejecutables que permiten realizar las operaciones básicas de ElGamal para el cifrado y descifrado de archivos. Damos a continuación el uso de dichas utilidades, con el objetivo de mostrar las capacidades de la implementación de ElGamal básico en PloneVoteCryptoLib:

- ***plonevote.gen_ cryptosys.py*** *–nbits=N –name="..." –description="..." filename*
  Esta utilidad genera una nueva instancia de ElGamal. Recibe como parámetros un nombre y una descripción para tal instancia, así como la longitud N de el primo p en bits. Finalmente, toma un nombre de archivo *filename*, en el cual guarda los detalles de la instancia de ElGamal como un documento XML (por convención, con extensión *.pvcryptosys*). Ese archivo contiene parámetros p y g para el esquema generados aleatoriamente dentro de los posibles valores correctos, así como el nombre y la descripción dados.

- ***plonevote.gen_ keys.py*** *–cryptosys=cryptosystem.pvcryptosys –private=private_ key.pvprivkey –public=public_ key.pvpubkey*
  Esta utilidad genera un nuevo par de llaves, dada una instancia de ElGamal (un archivo generado por la utilidad anterior) y los nombres de dos archivos donde guardar la llave pública y la llave privada, respectivamente. Las extensiones *.pvprivkey* y *.pvpubkey* son usadas por convención.

- ***plonevote.encrypt.py*** *–key=public_ key.pvpubkey –in=file.ext –out=file.pvencrypted*
  Esta instrucción cifra un archivo cualquiera *file.ext* con la llave pública *public_ key.pvpubkey* dada, guardando el archivo cifrado resultante en *file.pvencrypted*.

- ***plonevote.decrypt.py*** *–key=private_ key.pvprivkey –in=file.pvencrypted –out=file.ext*
  Esta instrucción descifra el archivo cifrado *file.pvencrypted* con la llave privada *private_ key.pvprivkey*, guardando el archivo descifrado en *file.ext*.

Las clases de python dentro de PloneVoteCryptoLib utilizadas por las utilidades anteriores serán descritas en la sección siguiente.

2.3.2. Arquitectura

Damos a continuación una breve descripción de la arquitectura interna e interfaz pública de nuestra implementación de ElGamal.

2.3.2.1. Clases de soporte: plonevotecryptolib.utilities

Antes de describir las clases principales de la implementación de ElGamal en PloneVoteCryptoLib, nos conviene explicar brevemente tres clases usadas como soporte para éstas que no forman parte de la biblioteca estándar de python. Estas clases han sido implementadas dentro del paquete plonevotecryptolib.utilities y son usadas ampliamente por nuestra biblioteca criptográfica.

- **BitStream**: La clase BitStream provee una forma estandarizada de tratar datos en PloneVoteCryptoLib como si se tratara de una secuencia uniforme de bits (e.g. "00111000101..."). Esta clase provee numerosos métodos para insertar números, cadenas de caracteres u otros datos en la secuencia de bits; así como métodos para recuperar los siguientes n bits, interpretados como cualquiera de esos tipos de datos. Los métodos de cifrado y descifrado de ElGamal operan, en su nivel más fundamental, sobre estas secuencia de bits. El uso de BitStream es la razón principal por la cual plonevote.encrypt.py y plonevote.decrypt.py pueden operar trivialmente sobre archivos binarios, al igual que de texto. Nuestra implementación de BitStream está basada en celdas, y es significativamente más eficiente en memoria que representar una secuencia de bits como un arreglo de enteros en python[17].

- **TaskMonitor**: TaskMonitor es un objeto que puede ser utilizado para monitorear el progreso de alguna operación larga dentro de PloneVoteCryptoLib. Múltiples métodos en nuestra biblioteca aceptan opcionalmente una instancia de TaskMonitor, a la cual informan cada cierto progreso realizado. Un cliente de PloneVoteCryptoLib, puede crear un TaskMonitor y registrar una o más funciones que deban ser llamadas cada vez que cierto progreso ocurra (representado como número de pasos o, cuando existe forma razonable de estimarlo, porcentaje de progreso). TaskMonitor puede ser utilizado por clientes para proveer indicadores o barras de progreso para las operaciones de PloneVoteCryptoLib. Para más información sobre sus capacidades completas, se recomienda ver la documentación en el propio archivo TaskMonitor.py. Este diseño es un ejemplo de los patrones de desarrollo de software: observador ("observer") y mediador ("mediator") [GHJV1995]; donde el sujeto observado es el progreso de algún método, los observadores son las funciones registradas con TaskMonitor y la clase en sí actúa como un mediador del protocolo entre estas dos entidades.

- **Enumerate**: Una implementación simple de enumeraciones (enums) para python.

2.3.2.2. Clases principales

La figura 2.3.1 muestra las clases involucradas con la implementación de ElGamal básico en PloneVoteCryptoLib, incluyendo sus relaciones y su interfaz pública[18]. La siguiente discusión explica los detalles más importantes de este diseño, así como las operaciones y formatos usados por nuestra implementación.

[17] 24x veces más eficiente en varias arquitecturas comunes.

[18] Las posibles excepciones lanzadas por algunos de los métodos no aparecen en el diagrama por razones de claridad, la documentación de los archivos correspondientes incluye esta información.

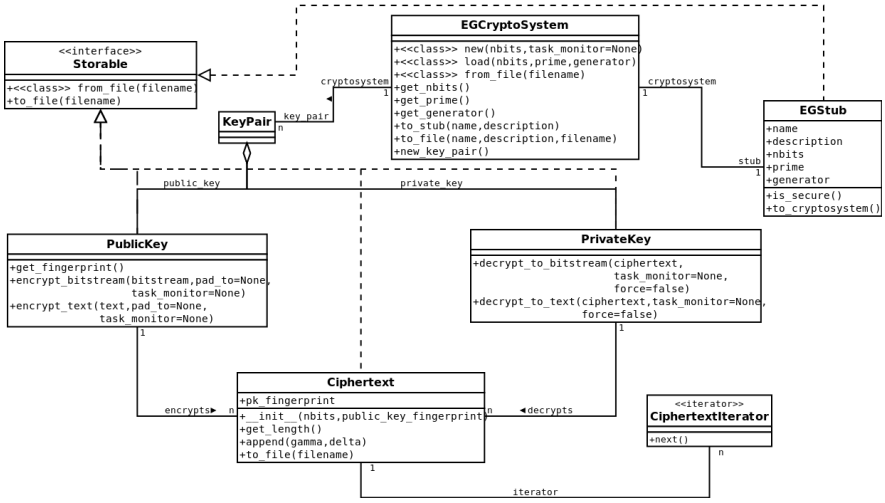

Figura 2.3.1: Diagrama de clases simplificado para ElGamal en PloneVoteCryptoLib

Storable es una interfaz[19] común a múltiples clases dentro de PloneVoteCryptoLib, proveyendo métodos para almacenar y recuperar su información desde un archivo XML. En un futuro, cuando PloneVoteCryptoLib haya sido más profundamente integrado con Plone, esta interfaz podría contener métodos para almacenar objetos de tal biblioteca como objetos en la base de datos de Zope, utilizada por Plone.

EGCryptoSystem representa una instancia del esquema ElGamal, lista para ser utilizada para generar un par de llaves. Los métodos *get_nbits()*, *get_prime()* y *get_generator()* permiten recuperar la longitud en bits de p y los valores de p y de g, respectivamente. El constructor de *EGCryptoSystem* es privado. Para obtener un objeto de esta clase alguno de los siguientes tres métodos de clase deben ser usados: *new*, el cual genera una nueva instancia del esquema ElGamal con un nuevo p aleatorio de longitud *nbits* dados (Alg. 2.1); *load*, el cual recibe y verifica los parámetros de la instancia de ElGamal directamente; y *from_file*, que intenta recuperar la instancia desde un archivo. En los tres casos, la clase *EGCryptoSystem* no permite la construcción de una instancia donde los parámetros dados no sean correctos, y los verifica antes de cargar un objeto de tal tipo mediante *load* o *from_file*. Por ejemplo, el primo p utilizado siempre debe ser un primo seguro, y g un generador del $\mathbb{Z}_p^*$ correspondiente. Esto nos garantiza[20] que siempre es seguro utilizar una

[19] Las interfaces como elementos del lenguaje de primera clase no son soportadas por python, debido a su sistema de tipos dinámico. Sin embargo, nos conviene pensar en que las operaciones listadas para *Storable* pertenecen a una interfaz compartida, para propósitos del diseño de PloneVoteCryptoLib.

[20] Salvo por cierta probabilidad de error del método de Miller-Rabin usado por pycrypto

Figura 2.3.2: Formato del mensaje en claro antes de ser cifrado.

instancia de *EGCryptoSystem* para generar un par de llaves.

La seguridad anterior, sin embargo, trae un costo: construir un nuevo objeto *EGCryptoSystem*, incluso al cargarlo desde un archivo, es significativamente caro. En algunos casos, quisiéramos rápidamente poder ver los parámetros de múltiples instancias de ElGamal antes de decidir cuál queremos usar. *EGStub* cubre ese caso de uso, dándonos una representación ligera (y sin verificar) de una instancia de ElGamal que no puede ser usada directamente para generar claves. Es posible convertir de *EGStub* a *EGCryptoSystem* y viceversa, mediante los métodos *to_cryptosystem* y *to_stub* respectivamente. Naturalmente, *to_cryptosystem* verifica los parámetros de *EGStub* antes de regresar el *EGCryptoSystem* correspondiente. El mismo formato de archivo es usado por ambas clases para guardar una instancia de ElGamal como archivo.

El método *new_key_pair* de *EGCryptoSystem* genera un nuevo par de claves dentro de esa instancia de ElGamal (Alg. 2.2), devolviendo un objeto de clase *KeyPair*. *KeyPair* simplemente es una agrupación de exactamente un objeto de clase *PublicKey* y otro de clase *PrivateKey*.

PublicKey es la clase usada para representar una llave pública dentro del esquema ElGamal. Su método *get_fingerprint* devuelve una huella digital (un hash SHA256) representando a dicha llave con bajo riesgo de colisión. Los métodos *encrypt_bitstream* y *encrypt_text* son usados, respectivamente, para cifrar un objeto BitStream o una cadena de caracteres con dicha llave pública, devolviendo un objeto de tipo *Ciphertext* (Alg. 2.4 por bloques).

Un detalle importante a notar es que los métodos de cifrado en *PublicKey* permiten, de forma opcional, dar un argumento *pad_to*. Dado dicho argumento, el texto claro es aumentado con datos aleatorios antes de ser cifrado, hasta que su tamaño completo sea, al menos, *pad_to* bytes. Como veremos en la sección 4.3.1.1, es importante que, en una elección en PloneVote, todos los votos emitidos sean del mismo tamaño; *pad_to* provee un mecanismo sencillo para garantizar lo anterior en la mayoría de las elecciones. Como es importante saber cuándo termina el mensaje real y comienzan los datos aleatorios, el tamaño del mensaje original es añadido al principio de éste antes de cifrarlo, independientemente de si se debe o no añadir algún relleno al final. El formato final de los datos antes de ser cifrados es el dado en la figura 2.3.2, el cual permite, una vez que el mensaje ha sido descifrado de nuevo, descartar los datos aleatorios añadidos.

La clase *PrivateKey* representa la llave privada del par, y sus métodos

para verificar primos de forma probabilística (ver 2.2.3.1).

decrypt_to_bitstream y *decrypt_to_text* son el inverso de los correspondientes en *PublicKey*, permitiendo descifrar un texto cifrado con la llave pública correspondiente, ya sea como un objeto BitStream o como una cadena de caracteres (Alg. 2.5 por bloques). Un detalle a mencionar es que, al cifrar, *PublicKey* incluye su propia huella digital codificada en el objeto *Ciphertext* resultante, la cual permite a *PrivateKey* saber si el texto fue cifrado con la llave pública correspondiente y abortar el descifrado en caso contrario (el parámetro *force* permite omitir esta comprobación).

Ciphertext es la clase usada para representar un texto cifrado. A diferencia de las clases anteriores, *Ciphertext* tiene un constructor público, que devuelve un nuevo texto cifrado vacío, compatible con cierta llave pública (el constructor recibe la longitud en bits de la instancia de ElGamal correspondiente y la huella digital de la llave pública). *pk_fingerprint* almacena la huella digital de la llave pública correspondiente a la instancia actual de *Ciphertext*, usualmente aquella correspondiente al objeto *PublicKey* que generó dicha instancia. Como mencionamos en la sección 2.1.1.2, debido a las operaciones que requiere PloneVote, debemos producir el texto cifrado como una extensión de ElGamal a cifrado por bloques (en vez de usar cifrado híbrido, por ejemplo), es decir, cada bloque de tamaño *nbits* del texto claro original es cifrado en *Ciphertext* como un par (γ, δ), donde cada componente del par tiene *nbits* de longitud. El método *append* de *Ciphertext* es usado únicamente por *PublicKey* para añadir bloques cifrados (γ, δ) al texto cifrado. Para lectura, *Ciphertext* es indizable e iterable, lo que quiere decir que la secuencia de pares (γ, δ) que lo forman puede ser accedida en python mediante la expresión *for* y/o el operador [ ].

2.3.2.3. params.py

El archivo params.py contiene algunos parámetros generales usados por PloneVoteCryptoLib. Estos parámetros incluyen, entre otros, el tamaño mínimo permitido en bits de las instancias de ElGamal que pueden ser generadas o cargadas dentro de la biblioteca, así como tamaños en bits por omisión. Aunque tales parámetros pueden ser variados a mano, editando los valores de variables de la forma *custom_propertyX*, la forma más natural de configurar PloneVoteCryptoLib es cambiando un solo parámetro: *params.SECURITY_LEVEL*, el cual controla los valores por omisión de los demás parámetros en la ausencia de especificaciones explícitas de la forma *custom_propertyX=val*.

Los valores posibles para *params.SECURITY_LEVEL* son aquellos valores incluidos en la enumeración *params.SECURITY_LEVELS_ENUM*. La tabla 2.2 muestra cada uno de dichos valores, junto con los valores que define para tres propiedades importantes: el tamaño por omisión de los esquemas ElGamal y llaves utilizadas, el tamaño mínimo permitido para dichos esquemas y llaves, y la probabilidad máxima de que los parámetros p o q no sean primos pero pasen las comprobaciones en *EGCryptoSystem.new*, *EG-*

CryptoSystem.load o *EGCryptoSystem.from_file*.

Nivel de seguridad	Tamaño de llaves por omisión	Tamaño de llaves mínimo permitido	Probabilidad de que p o q sean compuestos
INSECURE	128	0	$1/10^6$
LOWEST	1024	1024	$1/10^6$
LOW	2048	2048	$1/10^6$
NORMAL	4096	2048	$1/2^{128}$
HIGH	8192	3072	$1/2^{256}$
HIGHEST	15360	4096	$1/2^{256}$

Cuadro 2.2: Niveles de seguridad en PloneVoteCryptoLib

Capítulo 3

Cifrado de umbral

3.1. Cifrado de umbral en llave pública

3.1.1. El problema

En el capítulo 2 observamos a detalle el escenario en que un remitente busca usar un esquema de llave pública para enviar un mensaje de forma segura a un único destinatario, el cual puede recuperar el mensaje original del texto cifrado en cualquier momento. Para el protocolo de votaciones usado en PloneVote, vamos a necesitar un esquema ligeramente distinto: quisiéramos que cada mensaje requiriera de la intervención de múltiples usuarios, cada uno con su propia llave privada, antes de poder ser descifrado. Específicamente, quisiéramos que, dado un conjunto de n usuarios C y un valor $t \leq n$, hubiera un método para cifrar un texto claro cualquiera, de tal forma que cualquier subconjunto de t usuarios de C puedan colaborar para descifrar el mensaje cifrado, pero ningún subconjunto de C con menos de t usuarios pueda obtener información alguna sobre el mensaje original. Un esquema así con n destinatarios, de los cuales t son necesarios y suficientes para recuperar el mensaje original, se conoce como cifrado de umbral[1] t de n.

Como una primera aproximación a tal esquema, podemos observar lo siguiente:

1. Si $t = 1$, podemos cifrar el mensaje original m con cada una de las llaves públicas k_{e_i} de los n usuarios en C, concatenando los textos cifrados c_i resultantes. Cualquier usuario i en C podría entonces obtener el mensaje original de tal construcción, simplemente encontrando el texto cifrado c_i que le corresponde en la secuencia y descifrándolo con su llave privada. Un problema con este esquema es que requiere n rondas de cifrado para cada mensaje y aumenta el tamaño del mensaje cifrado

[1] *"Threshold encryption"* en inglés.

final n veces comparado con el resultado de cifrar m con una sola llave en el esquema de cifrado básico.

2. Si $t = n$, y k_{e_i} es la llave pública del i-ésimo usuario en C, podemos cifrar el mensaje por capas con cada llave pública como sigue (usando la notación para el esquema criptográfico dada en la sección 2.1): $c_1 = Enc(k_{e_1}, m)$, $c_i = Enc(k_{e_i}, c_{i-1})\ \forall i > 1$. Así, cada c_i es el resultado de cifrar c_{i-1} (o m si $i = 1$) con la llave pública del usuario i. Damos c_n como el texto cifrado final resultante. Dicho texto cifrado solo puede ser descifrado por el proceso inverso, en que cada usuario en C descifra en orden su correspondiente 'capa' del cifrado. Este esquema requiere, para el descifrado, que todos los usuarios actúen secuencialmente y en el orden correcto sobre el texto cifrado resultante, necesita de n rondas de cifrado para cada mensaje y, en el caso de ElGamal, aumenta el tamaño del texto cifrado resultante a $2^n |m|$, donde $|m|$ es el tamaño de m (pues la salida de cada operación de cifrado en ElGamal tiene dos veces la longitud de su entrada).

No solo son estos dos esquemas ineficientes en tiempo y espacio, sino que además no son los casos que nos interesan. El caso en que un solo usuario en C es capaz de descifrar el mensaje hace muy sencillo, como veremos en el capítulo 5, que cualquier usuario en C comprometa la privacidad de una elección en SVE o PloneVote. El caso en que $t = n$, en cambio, es susceptible a que un solo usuario de C se rehúse a cooperar, o simplemente extravíe su llave privada, impidiendo por completo el descifrado de los votos.

Uno puede imaginar un esquema en que cada mensaje m es cifrado por capas como en el segundo escenario para cada selección posible de t usuarios en C, y en que luego dichos textos cifrados sean concatenados para formar c, como se hizo con los textos cifrados para cada usuario en el primer escenario. Sin embargo, tal esquema es exponencial en tiempo y espacio en t, y sumamente impráctico de implementar.

De hecho, las operaciones elementales de un esquema de cifrado en abstracto no nos van a servir para implementar el esquema de cifrado de umbral requerido por PloneVote de forma eficiente. En la sección 3.2 veremos cómo utilizar algunas propiedades particulares del esquema ElGamal para construir el esquema de cifrado de umbral usado en PloneVote. Veremos que tal esquema no solo soporta cualquier elección de $t \leq n$, sino que además no aumenta el tamaño del texto cifrado c o el tiempo necesario para cifrarlo, comparado con el proceso de cifrado en ElGamal básico.

Usaremos el resto de esta sección para describir las propiedades del esquema de umbral de PloneVote, mediante aproximaciones sucesivas a dicho esquema, sin preocuparnos de momento sobre cómo son posibles algunas de las operaciones descritas.

Por razones que quedarán claras en la descripción del protocolo de elecciones en PloneVote dada en el capítulo 5, llamaremos en adelante al con-

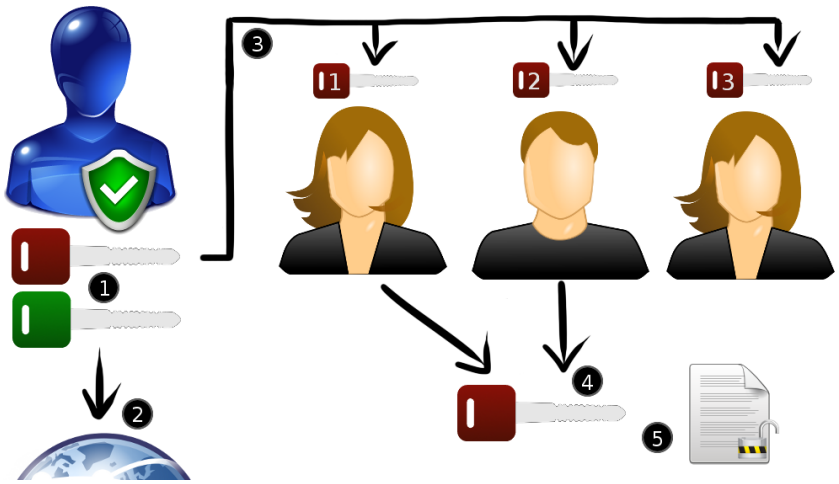

Figura 3.1.1: Compartición de llaves con umbral 2 de 3, usando un agente confiable.

junto C la comisión de la elección, y a los usuarios dentro de este conjunto miembros de la comisión.

3.1.2. Compartiendo llaves

Supongamos primero que, dado un cierto secreto, contamos con una forma de distribuirlo en n partes, de tal forma que t partes sean suficientes para recuperar el valor original del secreto, pero no exista forma alguna (o, por lo menos, ninguna forma eficiente en la práctica) de recuperar el secreto dadas $t-1$ partes de éste. En la sección 3.2.1 veremos un ejemplo de una forma tal de distribuir un secreto.

Si adicionalmente pudiéramos suponer que tenemos un agente confiable (una persona o una máquina que sabemos opera sin error y sin malicia), podríamos pensar en el protocolo mostrado en la figura 3.1.1 y descrito a continuación.

El agente confiable genera un par de llaves de nuestro esquema criptográfico (1). Dicho agente publica la llave pública a todos los posibles usuarios del sistema (2). Posteriormente, el agente parte la llave privada en n partes con el esquema de secreto compartido descrito anteriormente, para una cierta t dada, y distribuye cada una de tales partes a un miembro distinto de una comisión con n miembros (3).

Cuando se requiere descifrar un mensaje, t miembros de la comisión combinan sus partes de la llave privada para recuperar la llave privada original (4). Cualquiera de los t miembros involucrados posee ahora una copia de la llave privada y puede usarla para descifrar el mensaje (5), así como cualquier

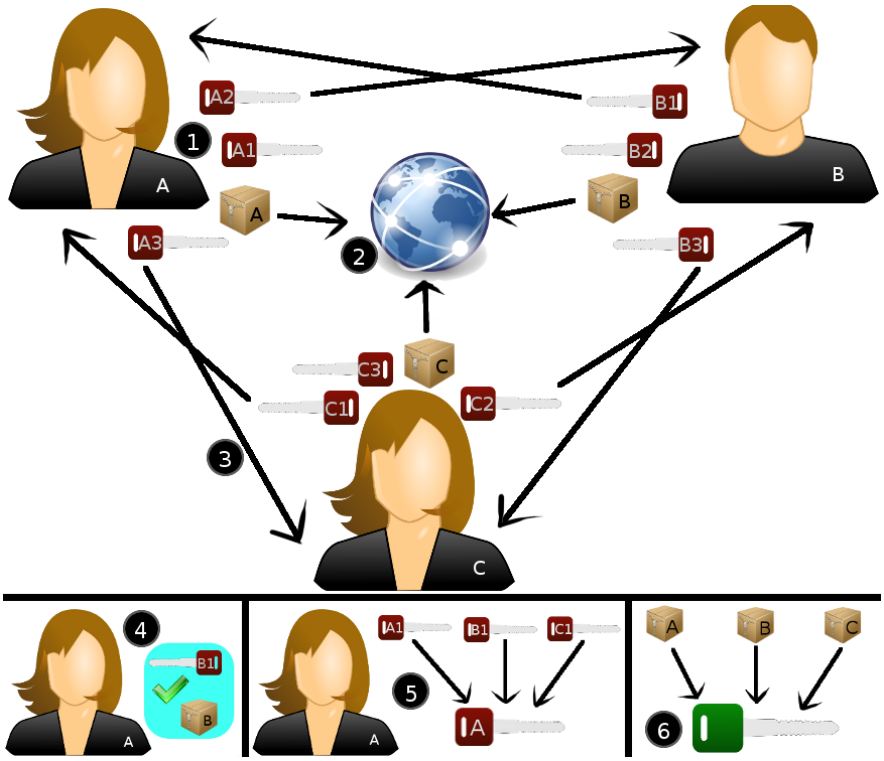

Figura 3.1.2: Compartición de llaves con umbral 2 de 3, mediante un protocolo distribuido seguro.

otro mensaje cifrado con la llave pública correspondiente.

El primer problema del esquema anterior es la dependencia en un agente confiable. En sistemas de votaciones tal agente nunca puede ser supuesto. De hecho, si un agente perfectamente confiable existiera, los usuarios podrían transmitir a éste de forma segura su voto y tal agente podría devolver los resultados finales de la elección, con total certeza para todos los involucrados.

Una alternativa que nos sería mucho más útil en nuestro escenario es que los miembros de la comisión pudieran ejecutar algún tipo de protocolo distribuido para generar la llave pública de la elección y sus llaves privadas correspondientes, en un esquema de umbral t de n, sin construir en ningún punto la llave privada general o dar información a ningún miembro sobre la llave privada de otro. Este protocolo existe, y será dado a detalle en la sección 3.2.2.2. De momento, la figura 3.1.2 muestra los pasos principales seguidos por los miembros de la comisión para ese esquema, y discutidos a continuación.

Cada miembro de la comisión genera un compromiso hacia la llave pú-

blica de umbral y una llave privada parcial para cada otro miembro de la comisión (1). Cada miembro publica su compromiso a todos los miembros de la comisión (2) y envía la llave privada parcial generada para cada otro miembro a éste, de forma privada (3). Cada miembro de la comisión revisa las llaves privadas parciales que recibe de otros miembros contra el compromiso publicado por el mismo remitente, verificando que ambos objetos correspondan a un par correctamente formado (4). Las llaves privadas parciales recibidas por cada remitente pueden ser combinadas eficientemente para generar su llave privada de umbral (5). Los compromisos publicados por todos los miembros de la comisión pueden, a su vez, ser combinados para generar la llave pública de umbral correspondiente (6).

Para descifrar un mensaje, t miembros de la comisión podrían combinar sus llaves privadas para obtener la llave privada general $\overline{k_d}$ correspondiente a la llave pública general $\overline{k_e}$. Cualquiera de los t miembros puede entonces usar la llave privada general para descifrar cualquier mensaje cifrado con la llave pública general correspondiente. Esto es análogo al paso (4) del escenario descrito en la figura 3.1.1.

3.1.3. Descifrado distribuido

Existe un segundo problema significativo en nuestro esquema original (figura 3.1.1) de compartir llaves, que desafortunadamente sigue presente en el esquema modificado mostrado en la figura 3.1.2. En ambos casos, cuando se requiere descifrar un mensaje cifrado en ese esquema, se debe recuperar la llave privada general. Una vez recuperada la llave, cualquiera de los t miembros de la comisión involucrados en obtenerla tiene acceso permanente a ésta y, por tanto, puede usarla para descifrar cualquier otro mensaje cifrado con la llave pública general correspondiente. Como veremos en el capítulo 5, lo que quisiéramos es que los miembros de la comisión (al menos t de ellos) tuvieran que estar de acuerdo para descifrar cada mensaje particular, sin que sea posible en ningún punto descifrar mensajes que no fueron elegidos para ser descifrados por la comisión como conjunto.

Lo que queremos, entonces, es un esquema de descifrado distribuido. Supongamos que los miembros de la comisión generan la llave pública de umbral y sus llaves privadas correspondientes, de forma distribuida, siguiendo el protocolo descrito en la figura 3.1.2. Quisiéramos ahora que fuera posible el escenario mostrado en la figura 3.1.3 y descrito a continuación.

Un subconjunto de t miembros de la comisión deciden cooperar para descifrar un texto cifrado c especifico (1). Cada miembro usa su propia llave privada, de forma independiente, para generar un "descifrado parcial" de c (2) y lo entrega a los demás miembros involucrados. Cualquier agente con acceso a t descifrados parciales del mismo c, puede obtener el mensaje original m ($c = Enc\left(\overline{k_e}, m\right)$), combinando dichos descifrados parciales (3), sin necesidad de conocer ninguna de las llaves privadas.

El escenario anterior puede ser aumentado, añadiendo una prueba a cada

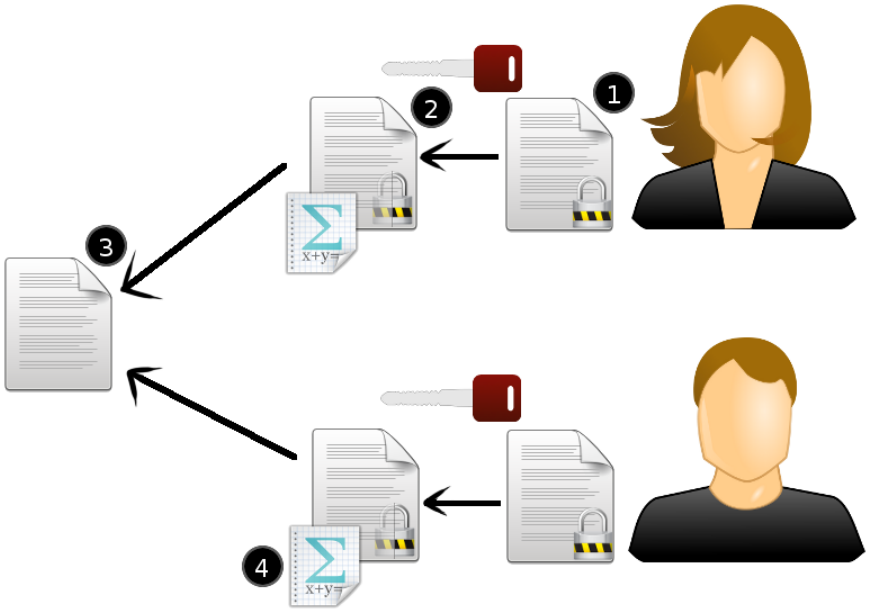

Figura 3.1.3: Descifrado distribuido en un esquema de umbral 2 de 3.

descifrado parcial (4), la cual puede ser usada para verificar que el descifrado parcial realmente corresponde al texto cifrado c.

Notemos que, en el escenario anterior, el contenido del mensaje descifrado es publicado (al menos dentro del grupo de t miembros de la comisión que cooperaron para descifrarlo). Sin embargo, la llave privada general $\overline{k_d}$ jamás es reconstruida o revelada a ningún usuario.

Nos importa, en tal esquema, que no sea posible obtener $\overline{k_d}$ a partir de ningún conjunto de descifrados parciales y que, además, sea imposible recuperar el mensaje m usando menos de t descifrados parciales del texto cifrado c correspondiente.

Tal vez sorprendentemente, el esquema descrito anteriormente para descifrado distribuido es posible y práctico de implementar, como se mostrará en la sección 3.2 (particularmente en 3.2.2).

3.2. Teoría: Cifrado de umbral y descifrado distribuido en ElGamal

3.2.1. Compartición de secreto de Shamir

El esquema de compartición de secreto de Shamir, *"Shamir Secret Sharing"* [Shamir1979], permite distribuir un secreto cualquiera con umbral t

de n. Esto es, permite dividir un secreto s en n partes, de tal modo que cualesquiera t partes sean suficientes para recuperar el secreto original. La siguiente definición formaliza las características de un esquema de compartición de secreto con umbral t de n (lo cual es distinto de un esquema de cifrado de umbral).

Definición 16. Esquema de compartición de secreto: Dado un secreto $s \in S$ (donde S es el conjunto de posibles secretos soportados por el esquema), un esquema de compartición de secreto con umbral t de n es un proceso mediante el cual s puede ser partido en n partes $s_1, ..., s_n$, cumpliendo las siguientes dos propiedades:

1. **Propiedad de correctitud**: El conocer cualesquiera t partes s_i distintas, permite calcular s de manera eficiente.

2. **Propiedad de seguridad débil**: Dadas a lo más $t-1$ partes s_i distintas, no existe ningún proceso para calcular s de manera eficiente (e.g. polinomial en $|s|$).

Podemos también pedir una propiedad de seguridad más poderosa a un esquema de compartición de secreto de umbral:

Definición 17. Propiedad de seguridad fuerte: Un esquema de compartición de secreto, cumple con la propiedad de seguridad fuerte, si y solo si, dadas a lo más $t-1$ partes s_i distintas, s queda completamente indeterminado. Es decir, dado cualquier $s' \in S$, no existe proceso alguno (eficiente o no), para distinguir si s' es o no el secreto representado por las partes s_i.

Es claro ver que la propiedad de seguridad fuerte implica la propiedad de seguridad débil.

El esquema de compartición de secreto de Shamir es un esquema de compartición de secreto que cumple con la propiedad de seguridad fuerte. Dicho esquema se basa en el siguiente resultado sobre polinomios en un campo:

Proposición 18. *Sea F un campo cualquiera y sean $(x_1, y_1), ..., (x_t, y_t) \in F \times F$, con x_i distintos. Existe entonces exactamente un polinomio $P(x)$ sobre F, de grado a lo más $t-1$, tal que $P(x_i) = y_i \ \forall i \in [1, t]$. Adicionalmente, todos los coeficientes de este polinomio pueden ser calculados eficientemente a partir de $(x_1, y_1), ..., (x_t, y_t)$.*

Demostración. Consideremos los coeficientes de Lagrange $\lambda_j(x)$, definidos a continuación como:

$$\lambda_j(x) = \prod_{l \in [1,t] \wedge l \neq j} \frac{x_l - x}{x_l - x_j}$$

Estos son polinomios en F, de grado a lo más $t-1$ (son multiplicación de $t-1$ términos de grado 1). Además, es fácil ver que cada coeficiente de Lagrange cumple que: $\lambda_j(x_j) = 1$ y $\forall l \in [1,t] \wedge l \neq j \ \lambda_j(x_l) = 0$.

Usando los coeficientes de Lagrange, podemos construir un polinomio P, como sigue:

$$P(x) = \sum_{j \in [1,t]} y_j \lambda_j(x)$$

Este polinomio es de grado a lo más $t-1$, por ser suma de polinomios de grado a lo más $t-1$. Por otro lado, $P(x_i) = \sum_{j \in [1,t]} y_j \lambda_j(x_i) = y_i \lambda_i(x_i) = y_i$, por las propiedades que vimos para los coeficientes de Lagrange. Adicionalmente, de nuestra construcción, podemos ver que obtener todos los coeficientes P es $O(t^2)$, por lo que sí podemos calcularlos de forma eficiente[2].

Falta tan solo mostrar que P es único.

Supongamos que existe otro polinomio $P'(x)$, de grado a lo más $t-1$, tal que $P'(x_i) = y_i \ \forall i \in [1,t]$. Tomamos $Q(x) = P(x) - P'(x)$, el cual tendrá también grado a lo más $t-1$, al ser suma de dos polinomios de grado a lo más $t-1$. Además, al ser P y P' distintos, Q no puede ser el polinomio cero. Pero entonces $Q(x_i) = P(x_i) - P'(x_i) = y_i - y_i = 0 \ \forall i \in [1,t]$. Al ser x_i distintas, tenemos que $Q(x)$ tiene t raíces distintas. Sin embargo, es un resultado clásico de álgebra (ver, por ejemplo, [Herstein1975, Lemma 5.2]), que un polinomio no cero de grado τ en un campo puede tener a lo más τ raíces distintas en tal campo. Esto nos da una contradicción, que sale de suponer que existen P y P' distintos con la forma pedida. Por lo tanto, P es único. □

Una forma intuitiva de expresar la proposición anterior es la siguiente: todo polinomio de grado $t-1$ en un campo se encuentra únicamente determinado por su valor en cualesquiera t puntos distintos. La proposición nos da, además, un algoritmo eficiente para obtener tal polinomio dados sus valores en t puntos.

En el esquema de compartición de secreto de Shamir con umbral t de n, S debe ser un campo, usualmente el campo finito $\mathbb{Z}_q$, donde $q > n$ es primo[3]. A continuación describiremos este esquema. Supondremos $S = \mathbb{Z}_q$ y todas las operaciones mencionadas serán implícitamente módulo q.

Dado el secreto $s \in S = \mathbb{Z}_q$ a compartir, se generan $a_1, ..., a_{t-1} \in \mathbb{Z}_q^*$ al azar, obteniendo el polinomio $P(x) = s + a_1 x + a_2 x^2 + ... + a_{t-1} x^{t-1}$ sobre $\mathbb{Z}_q$, de grado $t-1$. Cada parte $s_1, ..., s_n$ del secreto está dada por $s_i = P(i)$, mientras que $s = P(0)$. Recuperar s a partir de t partes s_i consiste en obtener

[2] En realidad, existen algoritmos aún más eficientes para obtener el polinomio que interpola t puntos, como aquellos dados en [Knuth1969] en $O(tlog^2 t)$. Sin embargo, $O(t^2)$ es suficientemente eficiente para nuestros propósitos.

[3] El uso de q en nuestra notación será aclarado en la sección 3.2.2.

Algoritmo 3.1 DividirSecreto(q,n,t,s)

1. Para $i = 1, ..., t-1$:
   a) Tomamos a_i al azar en $[1, q-1]$
2. Definimos $P(x) = s + a_1 x + a_2 x^2 + ... + a_{t-1} x^{t-1}$
3. Para $i = 1, ..., n$:
   a) Evaluamos $s_i = P(i)$
4. $s_1, ..., s_n$ son las partes del secreto s.

Algoritmo 3.2 RecuperarSecreto(q,n,t,$i_1, ..., i_t$,$s_{i_1}, ..., s_{i_t}$)

1. Para $i \in (i_1, ..., i_t)$:
   a) Calculamos $\lambda_i = \lambda_i(0) = \prod_{l \in (i_1,...,i_t) \wedge l \neq i} \left(\frac{l}{l-i}\right)$
2. Calculamos $s = P(0) = \sum_{i \in (i_1,...,i_t)} s_i \lambda_i$
3. Devolvemos el secreto compartido s

el polinomio $P(x)$ a partir de su valor en t puntos distintos $P(i) = s_i$, usando el cálculo de los coeficientes visto en la proposición 18.

Dos algoritmos describen el esquema en detalle: el algoritmo 3.1 describe la generación de las partes a partir de s, mientras que el algoritmo 3.2 da la recuperación de s desde t partes.

La propiedad de correctitud del esquema se sigue de la proposición 18. Dadas t parejas de la forma $(i, s_i) \in \mathbb{Z}_q \times \mathbb{Z}_q$, podemos (eficientemente) obtener un polinomio P' de grado a lo más $t-1$, que cumple $P'(i) = s_i$. Como tal polinomio es único, será el mismo $P(x) = s + a_1 x + a_2 x^2 + ... + a_{t-1} x^{t-1}$ usado para crear las partes. Por tanto, es trivial obtener s como $s = P(0) = P'(0)$.

La propiedad de seguridad fuerte también la podemos obtener de la proposición 18. Dadas $t-1$ parejas de la forma $(i, s_i) \in \mathbb{Z}_q \times \mathbb{Z}_q$, y cualquier $s' \in \mathbb{Z}_q$, añadimos la pareja $(0, s') \in \mathbb{Z}_q \times \mathbb{Z}_q$ a nuestro conjunto. Estas t parejas, por la proposición 18, nos definen algún polinomio P' tal que $P'(0) = s'$ y $P'(i) = s_i$. Por lo tanto, para cualquier $s' \in \mathbb{Z}_q$, las $t-1$ s_i dadas pudieron haber sido generadas como partes del secreto s', y no tenemos forma de distinguir s' cualquiera del secreto codificado por el conjunto de partes $s_1, ..., s_n$.

De lo anterior, tenemos que el esquema de compartición de secreto de Shamir es un esquema de compartición de secreto con umbral t de n, para

cualesquiera $t \leq n$ dadas, que cumple con la propiedad de seguridad fuerte.

El esquema de compartición de secreto de Shamir puede generalizarse trivialmente de $\mathbb{Z}_q$ a cualquier campo F. Sin embargo, una ventaja de usar $\mathbb{Z}_q$ es que cada una de las partes s_i es de tamaño a lo más $log\,(q)$, sin importar cuántas partes sean generadas. Si q es elegido de forma cuidadosa, las partes de s pueden tener el mismo tamaño que s (cada una), lo que no podríamos garantizar si tomáramos el campo $\mathbb{Q}$, por ejemplo. Adicionalmente, veremos en 3.2.2 que el esquema de compartición de secreto de Shamir en $\mathbb{Z}_q$ puede ser combinado con ElGamal para obtener un esquema de criptografía de umbral con las características particulares que buscamos (ver sección 3.1).

3.2.1.1. Linealidad del esquema de compartición de secreto de Shamir

Una propiedad importante del esquema de compartición de secreto de Shamir es que es lineal bajo suma, en el sentido de la proposición siguiente:

Proposición 19. *Sean s y v dos secretos compartidos, ambos dentro de un esquema de compartición de secreto de Shamir con umbral t de n y campo $\mathbb{Z}_q$, para algún q primo. Sean $s_1, ..., s_n$ las n partes de s y $v_1, ..., v_n$ las n partes de v. Entonces, $s_1 + v_1, ..., s_n + v_n$ son n partes, en el mismo esquema de compartición de secreto de Shamir (mismos t, n y q), del secreto compartido $s + v$.*

Demostración. Sea $P\,(x)$ el polinomio único de grado $t-1$ tal que $P\,(0) = s$ y $P\,(i) = s_i \ \forall i \in [1, n]$ (la unicidad se obtiene inmediatamente de la proposición 18 y $t \leq n$). Sea $Q\,(x)$ el polinomio único de grado $t-1$ tal que $Q\,(0) = v$ y $Q\,(i) = v_i \ \forall i \in [1, n]$. Tomemos $R = P + Q$, claramente de grado a lo más $t-1$ también. Por linealidad de la suma de polinomios: $R\,(x) = P\,(x) + Q\,(x) \ \forall x \in \mathbb{Z}_q$. En particular, $R\,(0) = s + v$ y $R\,(i) = s_i + v_i \ \forall i \in [1, n]$. Por construcción del esquema de compartición de secreto de Shamir, $s_1 + v_1, ..., s_n + v_n$ son n partes del secreto compartido $s + v$. □

Es importante notar que la suma a la que nos referimos aquí, al igual que durante nuestra descripción del esquema de compartición de secreto de Shamir, es la suma dentro del campo $\mathbb{Z}_q$. Es decir, la suma módulo q.

3.2.2. ElGamal de umbral y distribuido

3.2.2.1. Descifrado distribuido

Al discutir las características deseadas de un sistema de cifrado de umbral para PloneVote en la sección 3.1, pensamos primero en eliminar la necesidad de contar con un agente confiable para la generación de llaves y, posteriormente, consideramos la capacidad de realizar descifrado distribuido, sin revelar las llaves. En esta sección presentaremos la solución de estos problemas en forma inversa. Veremos a continuación cómo, suponiendo la

generación de llaves centralizada por un agente confiable A, podemos usar las llaves resultantes para realizar descifrado distribuido de umbral. En la sección 3.2.2.2, mostraremos cómo eliminar nuestra dependencia en el agente confiable, generando las llaves de forma distribuida y segura.

De momento, sea A nuestro agente confiable. A solo será necesario para generar las llaves del sistema, después de lo cual podemos retirarlo de nuestro escenario. Supongamos que contamos con n miembros de la comisión y buscamos que t sean necesarios para descifrar cualquier voto cifrado con la llave pública general de la elección. A genera u obtiene una instancia de ElGamal con primo seguro p y generador g. A genera entonces un par de llaves siguiendo el algoritmo 2.2. La llave pública generada $\overline{k_e} = g^a \ mod \ p$, será la llave pública general de la elección, y es publicada a todos los usuarios del sistema. Por otra parte, la llave privada $\overline{k_d} = a$ es dividida en n partes $k_{d1}, ..., k_{dn}$, usando un esquema de compartición de secreto de Shamir con umbral t de n y polinomio P, tal que $\overline{k_d} = P(0)$ y $k_{di} = P(i) \ \ \forall i \in [1, n]$. Cada parte k_{di} es entregada al miembro i de la comisión.

Dada la llave pública general de la elección $\overline{k_e}$, el proceso de cifrado es el mismo que en ElGamal básico (sección 2.2.4) usando esa llave. De lo que se sigue que la complejidad del proceso de cifrado o el tamaño del texto cifrado resultante no aumentan en nuestro esquema de cifrado de umbral, en comparación con ElGamal básico.

Notemos que A no entrega $\overline{k_d}$, la llave privada general de la elección, a ningún usuario. Esa llave solo existe como una función de las llaves privadas k_{di} de cada miembro de la comisión. Como vimos en la sección 3.2.1, t o más miembros de la comisión podrían cooperar para recuperar $\overline{k_d}$ a partir de sus respectivas k_{di}. Sin embargo, en vez de recuperar la llave privada general de la elección, quisiéramos que los miembros de la comisión cooperaran para descifrar un texto cifrado c generado por $Enc\left(\overline{k_e}, m\right)$, sin reconstruir $\overline{k_d}$ o facilitar a ninguno de los miembros involucrados el descifrado de un texto cifrado c', distinto de c.

Describiremos a continuación un protocolo para descifrado distribuido en el escenario anterior. El protocolo fue originalmente descrito por Cramer et al. [CGS1997]. Daremos de momento una versión simplificada del protocolo que aparece en ese artículo, sin adjuntar pruebas a los descifrados parciales; introduciremos las pruebas por separado en la sección 3.2.3.

Recordemos (sección 2.2.4) que

$$c = Enc\left(\overline{k_e}, m\right) = (\gamma, \delta) = (g^r \ mod \ p, \ m \cdot g^{ar} \ mod \ p)$$

para algún r elegido al azar. Supondremos en adelante todos los cálculos módulo p. Obtener m de $c = (\gamma, \delta)$ se reduce a obtener g^{ar} y dividir $m = \delta / g^{ar} = \delta \cdot g^{-ar}$.

Quisiéramos que t miembros de la comisión pudieran elevar g a la ar, de forma distribuida, sin reconstruir $a = \overline{k_d}$. Para lograr esto, cada uno de los miembros puede revelar (a los demás miembros involucrados) $z_i = \gamma^{k_{di}} =$

$g^{rk_{di}}$. Llamamos a cada z_i para un determinado c, un descifrado parcial de c. Bajo el supuesto de la dificultad del problema del logaritmo discreto, esto no revela k_{di} a ninguno de los otros miembros de la comisión, y los valores z_i no pueden ser utilizados para recuperar $\overline{k_d}$ (aun si r fuera también conocido). Sin embargo, t valores z_i para índices $i = \{i_1, ..., i_t\}$ distintos, sí pueden ser usados para obtener g^{ar}, mediante la fórmula siguiente:

$$z = \prod_{i \in \{i_1,...,i_t\}} z_i^{\lambda_i},$$

donde

$$\lambda_i = \lambda_i(0) = \prod_{l \in (i_1,...,i_t) \wedge l \neq i} \left(\frac{l}{l-i} \right).$$

Esto puesto que $a = \overline{k_d} = P(0) = \sum_{i \in (i_1,...,i_t)} k_{di}\lambda_i$ (ver sección 3.2.1, en particular el algoritmo 3.2), y entonces

$$\begin{aligned} z &= \prod_{i \in \{i_1,...,i_t\}} z_i^{\lambda_i} \\ &= \prod_{i \in \{i_1,...,i_t\}} \left(\gamma^{k_{di}}\right)^{\lambda_i} \\ &= \prod_{i \in \{i_1,...,i_t\}} \gamma^{k_{di}\lambda_i} \\ &= \gamma^{\sum_{i \in \{i_1,...,i_t\}} k_{di}\lambda_i} \\ &= \gamma^{\overline{k_d}} \\ &= \gamma^a \\ &= g^{ar}. \end{aligned}$$

Por tanto $z = g^{ar}$. Cualquier entidad que recibe al menos t z_i válidos y distintos, puede generar por su cuenta z, invertirlo en $\mathbb{Z}_p^*$ como $z^{p-2} = z^{-1} = g^{-ar}$ y obtener entonces $m = \delta \cdot g^{-ar}$.

Como mostramos que $z = g^{ar}$ este método de descifrado distribuido es correcto, provisto que los t participantes involucrados generen correctamente sus descifrados parciales z_i (veremos en la sección 3.2.3 como garantizar que los z_i son correctos). Adicionalmente, provista la dificultad del problema del logaritmo discreto, este método es seguro, en el sentido de que no revela la llave privada general. Pues si solo se conocen z_i y γ, no existe forma eficiente de recuperar $k_{di} = log_\gamma z_i$, y sin al menos t k_{di} distintos, no es posible recuperar $\overline{k_d}$. Finalmente, el único escenario en que conocer z permite descifrar un mensaje cifrado c', distinto de c, es cuando el valor aleatorio elegido durante el proceso de cifrado de c' es el mismo r que para c. Provista

una elección al azar de r en Enc y una instancia del esquema ElGamal de al menos 1024 bits, la probabilidad de tal colisión es menor que $\frac{1}{2^{1023}}$ [4].

3.2.2.2. Generación distribuida de llaves

Hemos mostrado que si un agente confiable A genera un par de llaves y otorga a los miembros de la comisión n partes $k_{d1}, ..., k_{dn}$ de la llave privada en un esquema de umbral t de n, entonces los miembros de la comisión pueden utilizar esas llaves privadas de umbral para descifrar mensajes de forma distribuida. Sin embargo, como vimos en 3.1, no podemos suponer la existencia de tal agente confiable para el escenario de elecciones en línea.

Buscamos en cambio un protocolo mediante el cual, dada una instancia (p, g) de ElGamal, n usuarios puedan generar, de forma distribuida y sin intervención de ningún otro usuario, $\overline{k_e} = g^a \ mod \ p$ y n partes $k_{d1}, ..., k_{dn}$ de $\overline{k_d} = a$ equivalentes a las que hubieran sido generadas al compartir a en el esquema de compartición de secreto de Shamir con umbral t de n. Buscamos además que, durante el protocolo, ningún usuario pueda conocer ningún k_{di} distinto al que le corresponde. Los usuarios externos (aquellos que no son uno de los n participando en el protocolo) no deberían recibir más información que $\overline{k_e}$ (es decir, no deberían obtener acceso a $\overline{k_d}$ o algún k_{di}). Finalmente, este protocolo debería ser capaz de resistir, o al menos detectar, ataques donde a lo más $t-1$ de los usuarios involucrados busquen corromper las llaves generadas.

El protocolo de Pedersen [Pedersen1991] para la generación distribuida de llaves en ElGamal cumple los requerimientos anteriores. El algoritmo 3.3 muestra una formulación de tal protocolo, donde $O_1, ..., O_n$ son los n usuarios involucrados (los miembros de la comisión en nuestro escenario), (p, g) son los parámetros de la instancia de ElGamal usada y $q = (p-1)/2$ es primo, pues p es un primo seguro.

La idea central del protocolo es la siguiente: cada usuario O_i genera su propio polinomio P_i de grado $t-1$ aleatorio. El polinomio $2P = 2\sum_{i\in[1,n]} P_i$ será entonces el polinomio del esquema de compartición de secreto de Shamir y, consecuentemente, $\overline{k_d} = 2P(0)$ será la llave privada general, $\overline{k_e} = g^{2P(0)}$ será la llave pública general y $k_{di} = 2P(i)$ será la llave privada de O_i. Durante la ejecución del procolo, P y $\overline{k_d}$ nunca son generados de forma explícita. Sin embargo, el protocolo publica $\overline{k_e}$ y permite a cada O_i obtener k_{di}.

Los polinomios P_i son generados en el campo $\mathbb{Z}_q$ y todas sus operaciones realizadas módulo q. Esto se debe a que los valores de los polinomios son siempre usados como exponentes en nuestras operaciones y $a = b \ mod \ (p-1) \Rightarrow x^a = x^b \ mod \ p \ \forall x \in \mathbb{N}$ [5]. Usamos q y no $p-1$ porque requerimos que q sea primo para que $\mathbb{Z}_q$ sea un campo, y que $\mathbb{Z}_q$ sea

[4] Equivalente a la probabilidad de descifrar $c' = \left(g^{r'} \ mod \ p, \ m' \cdot g^{ar'} \ mod \ p\right)$ adivinando r' al azar.

[5] Demostración: $a = b \ mod \ (p-1) \Rightarrow \exists k \ a = b + k(p-1) \Rightarrow x^a = x^{b+k(p-1)} =$

Algoritmo 3.3 Generación distribuida de llaves

1. Para $i \in [1,n]$:

   a) O_i genera al azar t coeficientes $a_{i0},...,a_{i(t-1)} \in \mathbb{Z}_q^*$. Estos coeficientes determinan un polinomio $P_i(x) = a_{i0} + a_{i1}x + ... + a_{i(t-1)}x^{t-1}$ de grado $t-1$.

   b) Para cada a_{il} $(l \in [0,t-1])$, O_i calcula $C_{il} = g^{a_{il}} \ mod \ p$ y lo publica entre todos los usuarios O_j.

   c) Para $j \in [1,n]$, O_i calcula $s_{ij} = P_i(j)$ y transmite s_{ij} de forma privada a O_j.

2. Todos los participantes esperan a haber recibido los s_{ij} y C_{il} de los demás participantes.

3. Para $i \in [1,n]$:

   a) Para cada $j \in [1,n]$, $i \neq j$, O_i verifica el valor s_{ji} recibido de O_j, revisando que $g^{2s_{ji}} = \prod_{l=0}^{t-1} (C_{jl})^{2^l}$. Si la verificación falla, la generación de llaves falla y O_i debe avisar a todos los demás participantes del problema.

   b) O_i calcula su llave privada como $k_{di} = 2s_i = 2\left(\sum_{j=1}^{n} s_{ji} \ mod \ q\right)$

   c) O_i calcula la llave pública general de la elección como $\overline{k_e} = \prod_{j=1}^{n} C_{j0}^2 \ mod \ p$ (recordemos $C_{j0} = g^{a_{j0}}$).

   d) Para cada $j \in [1,n]$, $i \neq j$, O_i calcula la llave pública de verificación para O_j, $v_j = \prod_{l=1}^{n} g^{2s_{lj}} \ mod \ p$.

un campo para que la interpolación de Lagrange, usada por el esquema de compartición de secreto de Shamir, sea posible. Si $a = b \ mod \ q$, entonces $2a = 2b \ mod \ (p-1)$ y $x^{2a} = x^{2b} \ mod \ p \ \forall x \in \mathbb{N}$. El usar q en vez de $2q = p-1$ requiere ajustar algunos cálculos, y es la razón de la presencia del número 2 como una constante en algunos pasos de nuestro algoritmo y en nuestra definición de $\overline{k_e}$, $\overline{k_d}$ y k_{di}.

Pedersen [Pedersen1991] define g no como un generador de $\mathbb{Z}_p^*$, sino como un generador del único subgrupo G_q de $\mathbb{Z}_p^*$ de orden q. Tal formulación le permite obtener directamente la propiedad de que $a = b \ mod \ q \Rightarrow g^a = g^b \ mod \ p$, y evita introducir la constante 2. Una desventaja de usar tal g, es que al combinar el esquema de Pedersen original con ElGamal (ver [Pedersen1991, Sección 4]), los mensajes m que pueden ser cifrados deben ser tomados dentro de G_q, lo cual lo hace menos práctico de implementar.

Notemos que si g es un generador de $\mathbb{Z}_p^*$, entonces g^2 es un generador de G_q. Cualquier elemento $\alpha \in G_q$ es un generador del subgrupo, pues G_q es un grupo de orden primo y el orden de cualquier elemento divide al orden del grupo, por lo que el orden de α es q, y por tanto α es un generador de G_q. Falta ver que $g^2 \in G_q$, pero esto es cierto, pues $\left(g^2\right)^q = g^{2q} = g^{p-1} = 1 \ mod \ p$, y $h^q = 1 \ mod \ p \Leftrightarrow h \in G_q$ ([Pedersen1991, Sección 2]).

El lector puede entonces ver que, en los cálculos del algoritmo 3.3, la constante 2 es usada para elevar al cuadrado una sola vez cada término completo de la forma g^x antes de compararlo módulo p con algún otro término. De hecho, podríamos reemplazar g por g^2 y omitir el resto de los usos de la constante 2, sin alterar la ejecución del algoritmo. Lo anterior nos daría el protocolo de Pedersen original, generando llaves que podríamos usar para cifrar mensajes $m \in G_q$ si ajustamos adecuadamente las funciones de cifrado y descifrado (Enc y Dec). Sin embargo, mantener la constante nos permite conservar la misma g y funciones Enc y Dec que usamos para el esquema ElGamal básico y refleja de modo más preciso nuestra implementación.

Mostraremos a continuación que nuestra variante del protocolo de Pedersen genera las llaves $\overline{k_e}$ y k_{di} necesarias para un esquema de cifrado de umbral en ElGamal (e.g. genera las mismas llaves que nos daba nuestro agente confiable en la sección 3.2.2.1), y protege adecuadamente la confidencialidad de las llaves k_{di} y $\overline{k_d}$. Estas dos propiedades son las que nos interesa obtener de nuestro protocolo de generación distribuida de llaves.

Veamos primero que la generación de las claves es correcta cuando todos los O_i se comportan de forma correcta. La llave privada k_{di} para O_i es correctamente generada en el paso ($3b$), pues

$$k_{di} = 2\left(\sum_{j=1}^{n} s_{ji} \ mod \ q\right) = 2\left(\sum_{j=1}^{n} P_j\left(i\right) \ mod \ q\right) = 2\left(\sum_{j=1}^{n} P_j\right)\left(i\right) = 2P\left(i\right).$$

$x^b \left(x^k\right)^{p-1} = x^b \cdot 1 \ mod \ p$.

Por otra parte, dados $C_{j0} = g^{a_{j0}} \ mod \ p \ \forall j$, con $P_j(x) = a_{j0} + a_{j1}x + ... + a_{j(t-1)}x^{t-1}$, tenemos

$$\begin{aligned}
\overline{k_e} &= \prod_{j=1}^{n} C_{j0}^2 \ mod \ p \\
&= \prod_{j=1}^{n} g^{2a_{j0}} \ mod \ p \\
&= g^{\sum_{j=1}^{n} 2a_{j0}} \ mod \ p \\
&= g^{\sum_{j=1}^{n} 2P_j(0)} \ mod \ p \\
&= g^{2\left(\sum_{j=1}^{n} P_j\right)(0)} \ mod \ p \\
&= g^{2P(0)} \ mod \ p.
\end{aligned}$$

Por lo cual, la llave pública es correctamente generada en el paso $(3c)$.

Lo anterior supone que los usuarios O_i siguen correctamente el protocolo de Pedersen. Un usuario malicioso O_j podría romper la relación entre k_{di} y $\overline{k_e}$, simplemente entregando coeficientes C_{j0} y elementos s_{ji} que no estuvieran relacionados por un polinomio común. La verificación del paso $(3a)$ asegura que la relación entre C_{j0} y s_{ji} es correcta.

$$\begin{aligned}
g^{2s_{ji}} &= \prod_{l=0}^{t-1} \left(C_{jl}\right)^{2i^l} \ mod \ p \\
&= \prod_{l=0}^{t-1} \left(g^{a_{jl}}\right)^{2i^l} \ mod \ p \\
&= \prod_{l=0}^{t-1} g^{2a_{jl}i^l} \ mod \ p \\
&= g^{2\sum_{l=0}^{t-1} a_{jl}i^l} \ mod \ p \\
&= g^{2P_j(i)}
\end{aligned}$$

En general, el revelar todos los coeficientes C_{jl}, compromete a O_j a usar un polinomio P_j particular, sin revelar dicho polinomio a los demás usuarios (suponiendo la dificultad del problema del logaritmo discreto). Fijado tal polinomio, la verificación del paso $(3a)$ garantiza que $s_{ji} = P_j(i)$, sin revelar P_j.

El último paso de nuestro algoritmo $(3d)$ genera las llaves públicas de verificación, las cuales son necesarias para verificar las pruebas de descifrado parcial (ver sección 3.2.3). Cada llave pública de verificación es $v_j = g^{k_{dj}} \ mod \ p$, pues

$$\begin{aligned} v_j &= \prod_{l=1}^{n} g^{2s_{lj}} \ mod\ p \\ &= g^{2\sum_{l=1}^{n} s_{lj}} \ mod\ p \\ &= g^{2\sum_{l=1}^{n} P_l(j)} \ mod\ p \\ &= g^{2P(j)} \ mod\ p \\ &= g^{k_{dj}} \ mod\ p. \end{aligned}$$

Suponiendo que O_i verificó s_{ji} en el paso $(3a)$, podemos obtener $g^{2s_{ji}} = \prod_{l=0}^{k-1} (C_{jl})^{2i^l}$, sin conocer s_{ji}.

Podemos combinar entonces el protocolo 3.3 con el protocolo de descifrado distribuido dado en la sección 3.2.2.1, para conseguir un esquema como el descrito en la sección 3.1. Nos falta tan solo resolver un problema, ¿cómo verificamos que los descifrados parciales z_i, generados por el usuario O_i, sean correctos y realmente correspondan al texto cifrado c que se busca descifrar?

3.2.3. Pruebas de descifrado parcial

3.2.3.1. Pruebas en conocimiento cero

El concepto de prueba en conocimiento cero (*Zero-Knowledge Proof*) fue introducido por Goldwasser, Micali y Rackoff en [GMR1985]. La idea fundamental es la siguiente: dado un demostrador P y un verificador V, un protocolo de prueba en conocimiento cero es un protocolo en que P y V interactúan, y mediante el cual P puede convencer a V de la validez de alguna proposición α, sin que V gane más información que la (probable) validez de α.

Algunos modelos consideran que P tiene capacidad de cómputo no-acotada, mientras que V es polinomialmente acotado. Nosotros consideraremos el caso en que tanto P como V son polinomialmente acotados, con la capacidad de tomar decisiones aleatorias, donde además P tiene acceso a alguna entrada adicional s que le permite demostrar α. P no puede transmitir s a V para demostrarle la validez de α, pues s da más información que la validez de tal propiedad. V y P deberán interactuar, durante un número polinomial de rondas y, al final de tal interacción:

- Si α es verdadera, P deberá ser capaz, con probabilidad 1, de convencer a V de ello.
- Si α es falsa, P no será capaz de convencer a V de que α es verdadera, salvo con una probabilidad muy baja que depende tan solo de las elecciones de V y no de las acciones de P.

Es importante notar que una prueba en conocimiento cero es un protocolo mediante el cual P convence a V de la veracidad de α, mas no constituye una

prueba matemática para α. En un protocolo de prueba en conocimiento cero, existe una cierta probabilidad de que P pueda "engañar" a V, haciéndole reconocer que la proposición α es verdadera cuando en realidad ésta es falsa. Sin embargo, en los protocolos de prueba en conocimiento cero que nos interesan, la probabilidad de que V acepte como verdadera una proposición α falsa es suficientemente pequeña como para considerarse inexistente en la práctica (e.g. $\rho \leq 2^{-128}$).

La forma usual de mostrar que ningún conocimiento adicional, más allá de la verdad de α, fue transmitido de P a V, consiste en dar un simulador C, que toma la misma entrada que V y supone α cierto. Si C puede ser dado, de tal modo que la distribución de sus salidas sea indistinguible de la distribución de las salidas de V interactuando con P (cuando α es cierto), incluyendo los mensajes transmitidos entre ambas entidades, entonces P es un demostrador en conocimiento cero de α.

Usualmente, los protocolos de prueba en conocimiento cero, para la mayoría de las propiedades α de interés, siguen el modelo siguiente:

1. P genera un compromiso I, que depende de s y de algún valor aleatorio generado por P, pero desde el cual no existe forma de recuperar s (al menos en tiempo polinomial).
2. P envía I a V
3. V envía un reto a P: φ
4. P da una respuesta a V que depende del reto φ
5. V verifica la respuesta de P, junto con el compromiso I, y usa esta información para ganar certeza de que α es cierta con alta probabilidad.
6. Si la respuesta y compromiso I de P concuerdan, V acepta α como verdadera.

En particular, la prueba en conocimiento cero para la igualdad de logaritmos discretos que daremos en la sección 3.2.3.2 sigue el esquema anterior.

Una descripción más formal del modelo de prueba en conocimiento cero puede ser encontrada en [GMR1985]. Una introducción detallada a estos protocolos puede ser encontrada en [Goldreich2002].

3.2.3.2. Prueba de conocimiento cero para descifrado parcial en ElGamal

En la sección 3.2.2.1 mostramos como múltiples miembros de la comisión pueden cooperar para, sin revelar sus llaves privadas, descifrar un cierto texto cifrado $c = (\gamma, \delta) = (g^r \ mod\ p,\ m \cdot g^{ar} \ mod\ p)$, generado de m como $c = Enc\left(\overline{k_e}, m\right)$, donde $\overline{k_e}$ es la llave pública general de la elección. Para realizar tal descifrado, cada miembro de la comisión calcula primero un descifrado parcial $z_i = \gamma^{k_{di}} \ mod\ p$, donde k_{di} es su llave privada. Posteriormente t

descifrados parciales distintos pueden ser combinados para obtener $z = g^{ar}$, mediante la fórmula siguiente:

$$z = \prod_{i \in \{i_1, \ldots, i_k\}} z_i^{\lambda_i},$$

el cual a su vez puede ser usado para recuperar m de δ.

Consideramos entonces que $t-1$ miembros de la comisión actuando maliciosamente no pueden descifrar c, u obtener la llave privada general de la elección $\overline{k_d}$. Sin embargo, un problema que omitimos es qué ocurre cuando algún miembro de la comisión genera un descifrado parcial $\bar{z_i}$ espurio, donde $\bar{z_i} \neq \gamma^{k_{di}} \ mod \ p$. Si esto ocurre, y posteriormente usamos tal $\bar{z_i}$ en nuestro proceso de combinación y descifrado, tanto z como $m' = \delta \cdot z^{-1}$ se verían alterados, y no contamos con ninguna garantía de que el resultado m' de tal proceso de "descifrado" fuera en realidad el mensaje m cifrado por c. De tal modo, un solo miembro malicioso de la comisión podría corromper el resultado del proceso de descifrado simplemente entregando un descifrado parcial espurio o incorrecto.

Para resolver el problema anterior, quisiéramos poder aumentar cada descifrado parcial con un certificado, el cual demuestre que $z_i = \gamma^{k_{di}} \ mod \ p$ (al menos con alta probabilidad), sin revelar k_{di}. Por supuesto, quisiéramos que tal certificado fuera verificable en tiempo polinomial.

Notemos que demostrar $z_i = \gamma^{k_{di}} \ mod \ p$, es equivalente a demostrar $log_g g^{k_{di}} = log_\gamma z_i$ (tomando $g^{k_{di}}$ y z_i como sus representantes módulo p). Conocemos $g^{k_{di}}$, pues es uno de valores de verificación v_i generados por el protocolo de generación distribuida de llaves dado en 3.3. Quisiéramos, entonces, que existiera una prueba en conocimiento cero (en particular, sin revelar información adicional sobre k_{di}), para mostrar que $log_g g^{k_{di}} = log_\gamma z_i$.

La prueba que buscamos es la prueba en conocimiento cero de la igualdad del logaritmo discreto, o prueba de Chaum-Pedersen [CP1992]. En dicha prueba, g y h son conocidos por un verificador V y un demostrador P, mientras que r es un secreto conocido solo por P. Dados $a = g^r \ mod \ p$ y $b = h^r \ mod \ p$, P puede demostrar a V que $log_g a = log_h b$ en conocimiento cero[6], mediante el protocolo 3.4. Suponemos de nuevo que $p = 2q + 1$, con p y q primos.

Notemos que la comprobación debe dar resultado verdadero cuando $log_g a = log_h b = r$ es cierto, pues

[6] En particular, sin revelar ninguna información que haga más sencillo para V calcular r, comparado con resolver el logaritmo discreto de $a = g^r \ mod \ p$.

Algoritmo 3.4 Prueba de Chaum-Pedersen

1. El demostrador P elije $s \in [1, p-2]$ al azar.
2. P calcula $I_1 = g^s \ mod\ p$ e $I_2 = h^s \ mod\ p$
3. P transmite I_1 e I_2 a V
4. V genera un reto $\varphi \in [1, p-2]$ al azar y lo transmite a P
5. P calcula $J = s + \varphi r \ mod\ (p-1)$ y transmite J a V
6. V verifica que $g^J = I_1 a^\varphi \ mod\ p$ y $h^J = I_2 b^\varphi \ mod\ p$, si esto es cierto, V acepta que $log_g a = log_h b$.

$$\begin{aligned} g^J &= g^{s+\varphi r \ (mod\ p-1)} \\ &= g^{s+\varphi r} \ mod\ p \\ &= g^s g^{\varphi r} \ mod\ p \\ &= I_1 (g^r)^\varphi \ mod\ p \\ &= I_1 a^\varphi \ mod\ p \end{aligned}$$

y análogamente para h^J.

Notemos que, si P pudiera conocer φ antes de generar I_1 e I_2, entonces podría elegir J al azar y generar I_1 e I_2 despejando de $g^J = I_1 a^\varphi \ mod\ p$ y $h^J = I_2 b^\varphi \ mod\ p$. De tal modo, P podría falsificar la prueba en conocimiento cero de $log_g a = log_h b$. La razón por la cual la probabilidad de que P convenza a V de que $log_g a = log_h b$, cuando esto es falso, sea muy baja, radica en que P entrega I_1 e I_2 a V antes de conocer φ. Para falsificar la prueba en conocimiento cero, P debería adivinar φ antes de que V lo seleccione (aleatoriamente). P tiene una probabilidad $\rho = 1/(p-1)$ de adivinar φ, con $p \geq 2^{1023}$ en una instancia de ElGamal de uso práctico. Por lo tanto, P solo puede engañar a V de este modo con muy baja probabilidad.

Para generar J de forma correcta, P debe conocer r. Si no lo conociera, pero pudiera obtener J de todos modos, entonces podría obtener r de $J = s + \varphi r \ mod\ (p-1)$. Por lo tanto, conocer J es equivalente a conocer r para P, dado que P conoce s, p y φ.

Por otro lado J no da información alguna a V sobre r. Para ver esto, notemos que, dados J y c cualquiera, podemos, para cualquier r, elegir s tal que $J = s + \varphi r \ mod\ (p-1)$. Por lo tanto, si V no conoce s, J y φ solos no le proporcionan información alguna sobre r. V solo puede conocer s resolviendo el problema del logaritmo discreto. El problema del logaritmo discreto no tiene solución eficiente y, además, si la tuviera, V podría obtener r directamente de $a = g^r \ mod\ p$, sin ayuda de P. De lo que podemos concluir

que P no da información alguna, más allá de que $log_g a = log_h b$ es cierto, a V.

3.2.3.3. La heurística de Fiat-Shamir

Una función de hash segura es una función $H : \{0,1\}^* \to \{0,1\}^m$, tal que no puede ser distinguida de una función aleatoria en tiempo polinomial. Más formalmente, una función H es una función de hash segura para nuestros propósitos si es resistente ante pre-imagen:

Definición 20. Resistencia ante pre-imagen: La función de hash $H : \{0,1\}^* \to \{0,1\}^m$ es resistente ante pre-imagen si, dada $y \in \{0,1\}^m$ cualquiera, no existe algoritmo polinomial alguno capaz de encontrar alguna x tal que $H(x) = y$, salvo con muy baja probabilidad.

Esencialmente, dada una función de hash segura (resistente ante pre-imagen) y un elemento y en su imagen, no existe un mejor método para encontrar x tal que $H(x) = y$, que el de probar aleatoriamente elementos $x' \in \{0,1\}^*$ y aplicarles H hasta encontrar alguno tal que $H(x') = y$. Podemos pensar, intuitivamente, que H es una función que, para cada entrada $x \in \{0,1\}^*$ que nosotros le demos, nos devuelve un elemento $y \in \{0,1\}^m$ elegido esencialmente al azar, con la única limitación de que H siempre responderá con la misma y para la misma x. Decimos entonces que H es una función pseudo-aleatoria.

La heurística de Fiat-Shamir [FS1986] es un método para convertir protocolos de prueba en conocimiento cero interactivos en algoritmos donde P genera por su cuenta una prueba de α, que puede ser verificada posteriormente por V sin intervención de P.

Notemos que, en el esquema general para pruebas de conocimiento cero dado en la sección 3.2.3.1, el cual es seguido por el protocolo de Chaum-Pedersen (Alg. 3.4), el rol de V se limita a dos partes: generar un reto φ una vez recibido el compromiso I y verificar la respuesta de P a ese reto. Si pudiéramos generar el reto φ sin intervención de V y sin que P pueda "elegir" φ o adivinarlo antes de generar su compromiso I, entonces solo necesitaríamos de V para verificar la respuesta final de P, y nos ahorramos el protocolo interactivo para demostrar α. Podemos generar tal reto φ, sin intervención de V, simplemente aplicando una función de hash segura H a los parámetros públicos de la prueba y al compromiso I de P. Por ejemplo, en el paso 4 del algoritmo 3.4, podemos reemplazar la actuación de V por P mismo generando $\varphi = H(a, b, I_1, I_2)$.

Como no existe forma de elegir la entrada de H para forzar una salida particular, esto es intuitivamente equivalente a que P reciba un reto φ al azar para cada compromiso I que genere. Esto es similar al caso en que P genera I, lo envía a V y recibe un reto al azar de V. Intuitivamente, estamos reemplazando V por H en ese paso del protocolo.

Una diferencia importante a notar al usar H para generar φ en vez de V, es que, en el caso no-interactivo, P podría repetir la generación de I y φ, hasta conseguir una pareja de valores (I, φ) que le sea "conveniente" (por ejemplo, uno para el cual puede falsificar el resultado de la prueba de α). Sin embargo, en la práctica P solo podrá repetir la generación de I y φ un número polinomial de veces y, si su probabilidad de engañar a V en el protocolo original era suficientemente pequeña, será tan solo polinomialmente mayor en el caso de la generación de prueba no interactiva. Para Chaum-Pedersen, en $\mathbb{Z}_p^*$ con p suficientemente grande, dicha probabilidad sigue siendo insignificante ($\rho < |p|^{c_1} / 2^{|p|}$).

El caso general del método anterior para convertir protocolos de prueba en conocimiento cero interactivos en algoritmos de generación de pruebas no-interactivos, usando una función de hash segura para construir el reto φ, es conocido como la heurística de Fiat-Shamir. Existen familias de funciones de hash que han sido demostradas resistentes ante pre-imagen en un sentido formal. Sin embargo, en la práctica, es más común usar funciones más simples y rápidas como SHA-2 [IETF2006], para las cuales no se conocen ataques de pre-imagen eficientes. Tal es la estrategia que empleamos en nuestra implementación.

3.2.3.4. Descifrado parcial con certificado

Podemos combinar la prueba de Chaum-Pedersen con la heurística de Fiat-Shamir para aumentar cada descifrado parcial generado por un miembro de la comisión con un certificado que muestre en conocimiento cero que $z_i = \gamma^{k_{di}} \ mod \ p$. El algoritmo 3.5 muestra cómo cada miembro de la comisión genera su descifrado parcial z_i, mientras que el algoritmo 3.6 describe el proceso necesario para combinar t descifrados parciales en un descifrado de $c = (\gamma, \delta) = (g^r \ mod \ p, \ m \cdot g^{ar} \ mod \ p)$, revisando antes los certificados de cada descifrado parcial. Los parámetros para ambos algoritmos son aquellos definidos en la sección 3.2.2.1, y se recomienda referirse a dicha sección para una descripción del funcionamiento de estos dos algoritmos, excepto por el uso de certificados (el cual sigue nuestra discusión de las secciones 3.2.3.2 y 3.2.3.3).

3.3. Implementación: Cifrado de umbral y descifrado distribuido en PloneVoteCryptoLib

3.3.1. Descripción y uso

PloneVoteCryptoLib implementa las funciones relacionadas con el cifrado de umbral en el paquete **plonevotecryptolib.Threshold**. Las clases de este paquete proveen las operaciones necesarias para generar las llaves de un esquema de ElGamal de umbral t de n, de forma distribuida y sin necesidad

Algoritmo 3.5 Construcción del descifrado parcial (para cada bloque $c = (\gamma, \delta)$)

1. O_i recupera el texto cifrado c para el cual busca generar su descifrado parcial z_i con prueba w_i
2. O_i genera $z_i = \gamma^{k_{di}} \ mod \ p$, usando su llave privada k_{di}
3. O_i elige $s \in [1, p-2]$ al azar, y genera $I_1 = g^s \ mod \ p$, $I_2 = \gamma^s \ mod \ p$
4. O_i genera el reto $\varphi = H\left(I_1, I_2, g^{k_{di}} \ mod \ p, z_i\right)$, donde H es una función de hash segura en la práctica
5. O_i toma $J = s + k_{di}\varphi \ mod \ (p-1)$
6. O_i publica (entre los miembros de la comisión) z_i y $w_i = (I_1, I_2, J)$

Algoritmo 3.6 Combinación de los descifrados parciales (para cada bloque $c = (\gamma, \delta)$)

1. Obtenemos k parejas (z_i, w_i), para $i \in \{i_1, ..., i_t\}$ distintos, correspondientes al mismo bloque de texto cifrado c
2. Para cada (z_i, w_i):
   a) Recuperamos I_1, I_2 y J de w_i
   b) Generamos el reto $\varphi = H\left(I_1, I_2, v_i, z_i\right)$, donde $v_i = g^{k_{di}} \ mod \ p$ (generado en Alg. 3.3)
   c) Verificamos $g^J = I_1\left(v_i^{\varphi}\right) \ mod \, p$ y $\gamma^J = I_2\left(z_i^{\varphi}\right) \ mod \, p$, rechazando el descifrado parcial en otro caso.
3. Para cada $i \in \{i_1, ..., i_t\}$:
   a) Calculamos el coeficiente de Lagrange

$$\lambda_i = \lambda_i\left(0\right) = \prod_{l \in (i_1, ..., i_t) \wedge l \neq i} \left(\frac{l}{l-i}\right)$$

4. Generamos $z = \prod_{i \in \{i_1, ..., i_t\}} z_i^{\lambda_i} \ mod \ p$
5. Recuperamos $m = \delta \cdot z^{-1} \ mod \ p$

de contar con ningún agente confiable, así como las operaciones necesarias para realizar descifrado distribuido con descifrados parciales públicamente verificables[7].

Durante la discusión anterior sobre cifrado de umbral en ElGamal, y sobre todo al hablar del protocolo distribuido de generación de llaves (sección 3.2.2.2), supusimos que los miembros de la comisión podían comunicarse de forma privada, además de publicar información visible por toda la comisión. En nuestra implementación final de PloneVote, los usuarios solo se comunican directamente con un servidor central, utilizando dicho servidor para retransmitir mensajes entre ellos. Durante esta interacción, no debe considerarse al servidor como un agente confiable (de nuevo, si contáramos con un agente confiable, el protocolo de votaciones es trivial). Esto es, el servidor puede leer, modificar y re-transmitir a usuarios no autorizados los mensajes que éste reciba, así como generar mensajes espurios como respuesta a las peticiones de usuarios legítimos. Debemos entonces implementar dos tipos de canales sobre un servidor no-confiable:

- **Foro de mensajes** (*bulletin board*): Deberá ser posible publicar un mensaje M en el servidor de tal forma que cualquier usuario pueda leer dicho mensaje y éste no sea alterado una vez publicado. Este escenario parece trivial, pero implica el requerimiento de que cualquier usuario que solicite M reciba exactamente el mismo mensaje y éste sea aquel publicado por el usuario original que generó M. Podemos verificar lo anterior mediante el uso de huellas digitales de los mensajes, obtenidas aplicando una función de hash seguro a M. Veremos cuándo y cómo son revisadas estas huellas digitales cuando discutamos las etapas del subprotocolo de cifrado de umbral de PloneVote. De momento, consideraremos que la acción de "publicar" un mensaje en el servidor implica transmitirlo al servidor y que éste devolverá el mismo mensaje a cualquier usuario que lo solicite[8].

- **Canales privados virtuales**: Deberá ser posible a un usuario A transmitir un mensaje M a un usuario B, a través del servidor, sin que dicho mensaje pueda ser leído por ningún usuario distinto de B. Si este escenario nos resulta familiar, es porque es esencialmente aquel resuelto por criptografía estándar de llave pública (sección 2.1.2). Si A tiene acceso a la llave pública $B.k_e$ de B, éste podrá simplemente cifrar M con dicha llave pública y publicar el mensaje cifrado $C = Enc\,(B.k_e, M)$ en el servidor (en modo de foro de mensajes). Aunque cualquier usuario

[7] Esto es, cualquier entidad con acceso al descifrado parcial, puede verificar la prueba de descifrado parcial adjunta.

[8] En la práctica, el servidor puede denegar acceso al mensaje M a usuarios que no tengan un motivo válido para acceder a éste (por ejemplo, personas que no son ni miembros de la comisión, ni votantes, ni auditores en la elección). Sin embargo, no podemos contar con ello para la seguridad del protocolo, pues un servidor malicioso siempre puede filtrar cualquier información a la cual tiene acceso, a usuarios no autorizados.

podría en teoría recuperar C del servidor, solo B puede descifrarlo para recuperar M. Esto garantiza la privacidad de los mensajes enviados de esta forma. La integridad de los mensajes, esto es, la certeza de que C (y por tanto M) no fue alterado por el servidor, podría ser garantizada mediante firmas digitales. Sin embargo, nuestra implementación actual hace uso, en cambio, del mismo sistema de huellas digitales usado para publicar mensajes en modo de foro de mensajes[9].

Podemos ver la parte del protocolo de PloneVote que se relaciona directamente con el cifrado de umbral, como un proceso en cuatro fases. Nos referiremos a este proceso como el "subprotocolo de cifrado de umbral" de PloneVote, y lo usaremos en lo que resta de esta sección para hablar de los servicios ofrecidos por **plonevotecryptolib.Threshold**, en el contexto de su uso esperado. Veremos la descripción del protocolo completo de votaciones usado por PloneVote en la sección 5.2.3.

Las fases del subprotocolo de cifrado de umbral son las siguientes:

1. **Establecimiento de los parámetros básicos**: Esta fase es realizada cuando se configura la elección en PloneVote. Durante esta fase se elige un número n y n usuarios $O_1, ..., O_n$ que actuarán como miembros de la comisión, junto con el umbral t de éstos que será necesario para descifrar los votos. Se eligen también en esta fase los parámetros (p, g) de una instancia de ElGamal válida, que será usada para generar las llaves utilizadas para el cifrado de umbral. Todos los parámetros anteriores pueden ser decididos por un administrador de la elección designado, y son públicos.

2. **Establecimiento de canales privados virtuales**: Cada miembro de la comisión O_i genera un par de llaves de ElGamal básico y publica su llave pública en el servidor. Para mayor seguridad O_i puede usar un canal externo al sistema (e.g. correo electrónico o en persona) para comunicar a los demás miembros de la comisión la huella digital de su llave pública, permitiendo verificarla. Notemos que la llave pública puede ser generada dentro de una instancia distinta de ElGamal que la usada para el cifrado de umbral, siempre y cuando los parámetros sean correctos y suficientemente seguros (p suficientemente grande). Adicionalmente, una vez que O_i publica su llave pública en el servidor, éste podría usar el mismo par de llaves para establecer canales privados virtuales en elecciones subsecuentes, omitiendo esta fase, siempre y cuando la llave privada no sea comprometida.

[9] Las llaves públicas usadas para establecer canales privados virtuales son publicadas en el servidor en modo de foro de mensajes. Si usáramos firmas digitales, estas mismas llaves serían usadas para verificarlas. Sin embargo, la seguridad aún dependería del uso de huellas digitales para verificar las llaves públicas en sí. Usar huellas digitales para comprobar la integridad de todos los mensajes simplifica nuestra implementación y protocolo, y usar firmas digitales no nos proporciona una mayor seguridad en este escenario.

3. **Establecimiento del esquema de umbral**: En esta fase, los miembros de la comisión realizan una variante del protocolo 3.3 de generación distribuida de llaves para el esquema de umbral, en la cual la comunicación privada entre usuarios es reemplazada por el uso de nuestros canales privados virtuales. Esta fase se realiza en dos rondas:

   a) **Generación de compromisos**: Cada miembro de la comisión O_i genera su polinomio secreto y un compromiso $\overline{C_i}$ que incluye los coeficientes públicos $C_{ij} = g^{a_{ij}}$, así como los valores s_{ij} de su contribución a la llave privada de umbral de cada otro miembro O_j (ver Alg. 3.3), cifrados con la llave privada básica de O_j. Es decir, s_{ij} es transmitido a O_j por un canal privado virtual, mientras que los coeficientes públicos son publicados en modo de foro de mensajes, aun cuando todos estos valores son agregados como el compromiso $\overline{C_i}$ de O_i.

   b) **Generación de llaves**: Cada miembro de la comisión O_i descarga los compromisos de los demás miembros y los junta con una copia local del suyo propio para generar el conjunto de compromisos C. O_i genera una huella digital de C, y debería compararla, por un canal externo al sistema, con aquella obtenida por otros miembros de la comisión, para asegurar que todos tienen el mismo conjunto de compromisos y que cada $\overline{C_j}$ fue generado por su respectivo O_j. O_i realiza entonces la segunda parte del protocolo 3.3 y genera su llave privada de umbral k_{di}, así como la llave pública general de la elección k_e. k_e es publicada en el servidor. Para mayor seguridad, los miembros de la comisión deben distribuir una copia de la huella digital de k_e a los votantes, por un canal externo al sistema.

4. **Descifrado distribuido**: Entre el paso anterior y éste ocurre la votación y el proceso de mezcla de votos descrito en el capítulo 4. Una vez listo el conjunto de votos que debe ser descifrado, cada miembro de la comisión descarga esos votos y genera su descifrado parcial verificable siguiendo el algoritmo 3.5. Posteriormente, cualquier usuario puede descargar estos descifrados parciales, verificarlos[10] y combinarlos para obtener el conjunto de votos descifrados (Alg. 3.6), a partir de los cuales puede realizar el conteo de la elección. Por conveniencia, el servidor podría publicar su propio conteo en cuanto haya al menos t descifrados parciales correctos publicados.

El subprotocolo anterior describe el uso esperado de la funcionalidad provista por **plonevotecryptolib.Threshold** en PloneVote. Dado que dicha

[10] Sin necesidad de huellas digitales o confiar en los miembros de la comisión, simplemente usando la prueba de equivalencia del logaritmo discreto adjunta al descifrado parcial (ver sección 3.2.3).

funcionalidad depende íntimamente de la capacidad de interactuar con un servidor (o directamente con un conjunto de otros usuarios), PloneVoteCryptoLib no provee ninguna herramienta en **plonevotecryptolib.tools** demostrando esta funcionalidad.

3.3.2. Arquitectura

La figura 3.3.1 muestra las clases principales dentro de PloneVoteCryptoLib involucradas en dar soporte al esquema de cifrado de umbral y al subprotocolo resultante descrito en la sección anterior, junto con sus métodos, atributos y relaciones más importantes. Al igual que en la sección 2.3.2, omitimos alguna información no esencial de nuestro esquema, en particular aquella referente a las excepciones que pueden ser lanzadas por algunos métodos cuando sus parámetros son incorrectos o fallan las verificaciones de seguridad. Salvo por aquellas marcadas en gris[11], todas estas clases forman parte del paquete **plonevotecryptolib.Threshold**.

Varias de las clases descritas implementan la interfaz **Storable**, que permite a éstas ser codificadas como archivos y publicadas en el servidor. Para más información sobre la interfaz **Storable**, el lector puede referirse a la sección 2.3.2.

Una clase omitida en este diagrama es **CoefficientsPolynomial**, la cual es una clase de soporte que nos permite generar polinomios con coeficientes aleatorios en $\mathbb{Z}_q$ para un q dado, y evaluar su valor módulo q en cualquier punto. **ThresholdEncryptionSetUp** utiliza esta clase para generar compromisos.

A continuación describimos en orden cada una de las clases en la figura 3.3.1, enfocándonos en su papel dentro del subprotocolo de cifrado de umbral de PloneVote (ver sección 3.3.1).

La clase *ThresholdEncryptionSetUp* se encarga de mediar la fase de establecimiento del esquema de umbral del subprotocolo descrito en la sección 3.3.1, encapsulando las operaciones requeridas por el algoritmo 3.3, para la generación distribuida de llaves para el esquema de umbral. El constructor de *ThresholdEncryptionSetUp* toma la instancia de ElGamal que será utilizada para generar las llaves del esquema de umbral (como un objeto *EGCryptoSystem*), el número total n de miembros de la comisión (*num_trustees*) y el valor del umbral t (*threshold*).

Para la generación de compromisos (fase $3a$ del subprotocolo de cifrado de umbral), cada miembro O_i de la comisión debe primero proporcionar las llaves públicas de todos los miembros de la comisión a una instancia adecuada de *ThresholdEncryptionSetUp*, mediante el método *add_trustee_public_key*. Este método toma el número que identifica al miembro correspondiente de la comisión, junto con su llave pública. O_i puede tomar las llaves públicas de

[11] Las cuales fueron descritas en 2.3.2, y se mencionan aquí en términos de su relación con las clases de cifrado de umbral introducidas en esta sección.

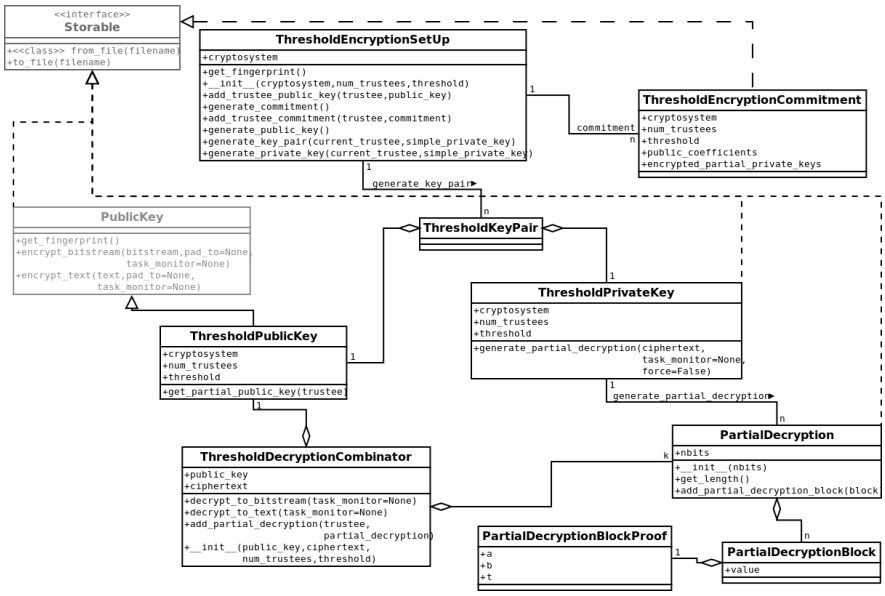

Figura 3.3.1: Diagrama de clases simplificado para ElGamal de umbral en PloneVoteCryptoLib

otros miembros de la comisión del servidor, verificando que sus huellas digitales correspondan con aquellas comunicadas por los respectivos miembros mediante canales externos al sistema.

Una vez cargadas las llaves públicas, cada miembro de la comisión puede usar el método *ThresholdEncryptionSetUp.generate_commitment()* para construir un compromiso nuevo de acuerdo con la primera parte del algoritmo 3.3. Cada compromiso es un objeto de clase *ThresholdEncryptionCommitment*. El cual contiene los coeficientes públicos y llaves privadas parciales cifradas para cada otro miembro de la comisión. Tras generar su compromiso, O_i deberá publicarlo en el servidor, completando la fase $3a$ del subprotocolo de cifrado de umbral.

La figura 3.3.2 muestra un diagrama de interacción de clases que describe el proceso anterior para la generación de compromisos. Las entidades "[Cliente]" y "[Servidor]", representan al programa cliente que utiliza PloneVoteCryptoLib y al servidor de PloneVote respectivamente. Estas entidades y sus mensajes no están implementadas dentro de nuestra biblioteca, sino que modelan entidades externas a ésta. Las otras dos entidades se mapean directamente a las clases del mismo nombre que acabamos de discutir, y sus mensajes a los métodos correspondientes.

Una vez que todos los miembros de la comisión hayan generado y publicado sus compromisos, la clase *ThresholdEncryptionSetUp* es usada de nuevo, ahora para generar el par de llaves de umbral para cada miembro

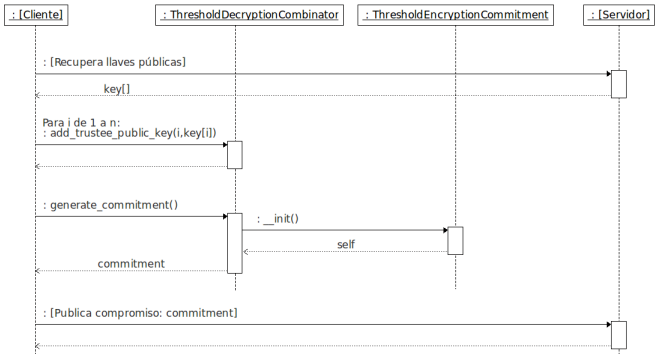

Figura 3.3.2: Diagrama de interacción de clases para generación de compromisos

de la comisión, de acuerdo con la fase $3b$ del subprotocolo de cifrado de umbral. Cada O_i vuelve a construir un objeto de tipo *ThresholdEncryptionSetUp* con los parámetros apropiados (o vuelve a usar el construido anteriormente, si tal objeto está aún disponible) y registra ahora con éste los compromisos de cada miembro de la comisión[12]. El método *ThresholdEncryptionSetUp.add_trustee_commitment* cubre este propósito, con argumentos análogos a los de *add_trustee_public_key*. Es muy importante en este caso que O_i proporcione su propio compromiso desde una copia local en almacenamiento seguro, usando los compromisos publicados en el servidor solo para los demás miembros de la comisión.

Una vez cargados todos los compromisos, O_i puede invocar el método *ThresholdEncryptionSetUp.get_fingerprint()*, el cual devuelve una huella digital de la colección completa de compromisos. O_i debe distribuir tal huella digital a los otros miembros de la comisión por un canal externo al sistema. Si cada O_i carga su propio compromiso de almacenamiento seguro y las huellas digitales de la colección de compromisos vistas por todos los miembros coinciden, entonces éstos pueden confiar en que están usando todos el mismo conjunto de compromisos y que éstos son realmente aquellos que cada uno generó en la fase de generación de compromisos.

Una vez cargado y verificado el conjunto de compromisos, cada O_i puede obtener su propio par de llaves de umbral, mediante el método *ThresholdEncryptionSetUp.generate_key_pair*. Tal método requiere el índice de O_i dentro de la comisión, así como su llave privada básica, necesaria para descifrar las llaves privadas parciales cifradas para éste por los otros miembros de la comisión en sus compromisos. *ThresholdEncryptionSetUp.generate_key_pair* devuelve una instancia de la clase *ThresholdKeyPair*, la cual simplemente agrupa una instancia de *ThresholdPrivateKey* y otra de *ThresholdPublicKey*.

[12] No es necesario volver a registrar las llaves públicas básicas de los miembros de la comisión.

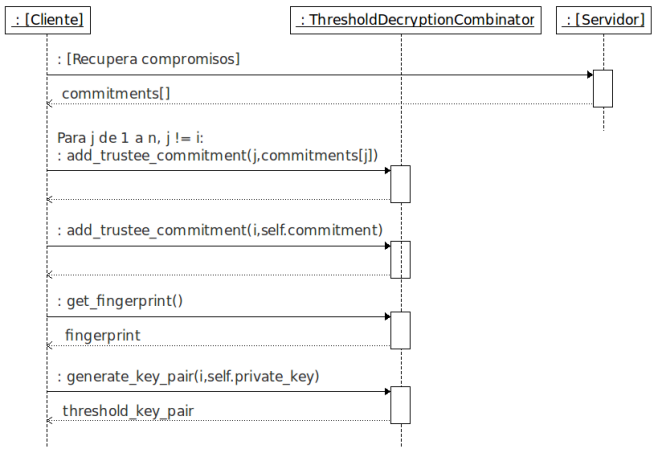

Figura 3.3.3: Diagrama de interacción de clases para generación de par de llaves de umbral

La figura 3.3.3 muestra un diagrama de interacción correspondiente a la generación del par de llaves de umbral para un miembro de la comisión O_i, utilizando *ThresholdEncryptionSetUp*.

ThresholdEncryptionSetUp permite también generar la llave privada de umbral de un miembro de la comisión por sí sola, mediante el método *ThresholdEncryptionSetUp.generate_private_key*, así como solo la llave pública de umbral general, mediante el método *ThresholdEncryptionSetUp.generate_public_key*. Adicionalmente, *generate_public_key* no requiere de la llave privada básica de ningún miembro de la comisión, permitiendo que cualquier entidad o usuario, con acceso a los compromisos de todos los miembros de la comisión, construya la llave pública de umbral. En PloneVote, el propio servidor podría, por conveniencia, realizar esta acción[13].

ThresholdPublicKey representa a la llave pública de umbral o llave pública general de la elección. Esta clase hereda de *PublicKey* (ver sección 2.3.2) y presenta una interfaz compatible con tal superclase. Esto es, cualquier programa que utiliza una llave pública básica de PloneVoteCryptoLib para cifrar mensajes, es trivialmente capaz de usar una llave pública de umbral[14]. Adicionalmente, *ThresholdPublicKey* incluye como atributos las propiedades n y t del esquema de umbral al que corresponde, y provee el método *get_partial_public_key*, el cual permite recuperar la llave pública parcial o valor de verificación $v_i = g^{k_{di}}$ para cada miembro de la comisión (ver Alg.

[13] Para garantizar la seguridad de la elección, sería necesario, sin embargo, verificar que la huella digital de la llave pública de umbral generada por el servidor corresponda a la generada independientemente por los miembros de la comisión, y posiblemente uno o más auditores.

[14] Esto incluye plonevote.encrypt.py dentro de plonevotecryptolib.tools.

3.3). Las llaves públicas parciales v_i son utilizadas internamente por la clase *ThresholdDecryptionCombinator*, descrita más adelante, para verificar las pruebas de descifrado parcial.

A diferencia de *ThresholdPublicKey*, *ThresholdPrivateKey* no es compatible con las llaves privadas básicas de PloneVoteCryptoLib, puesto que su interfaz es significativamente distinta. Esta clase representa una llave privada de umbral, y no provee métodos para descifrar, por sí sola, un texto cifrado con la llave pública de umbral. En cambio, provee el método *ThresholdPrivateKey.generate_partial_decryption* que genera un descifrado parcial dado un texto cifrado con la llave pública de umbral correspondiente.

Los descifrados parciales son representados por la clase *PartialDecryption*, la cual es una colección indizable de bloques *PartialDecryptionBlock*. Cada bloque de estos contiene el valor del descifrado parcial z_i de un bloque $c = (\gamma, \delta)$ del texto cifrado a partir del cual fue generado el descifrado parcial. Cada bloque incluye también un objeto de tipo *PartialDecryptionBlockProof*, el cual encapsula los valores de la prueba de descifrado parcial w_i correspondiente (ver Alg. 3.5, los atributos a, b y t de *PartialDecryptionBlockProof* corresponden a los valores I_1, I_2 y J del algoritmo, respectivamente).

Finalmente, *ThresholdDecryptionCombinator* es una clase que encapsula el algoritmo de combinación de descifrados parciales (Alg. 3.6). Su constructor toma la llave pública de umbral correspondiente al texto a descifrar, el texto cifrado en sí y los parámetros n (*num_trustees*) y t (*threshold*) del algoritmo de umbral. Una vez construida una instancia, es necesario proporcionar a esta clase al menos t descifrados parciales, mediante el método *ThresholdDecryptionCombinator.add_partial_decryption*, junto con el índice del miembro de la comisión al que éste corresponde. Ya registrados suficientes descifrados parciales, los métodos *decrypt_to_bitstream* y *decrypt_to_text* pueden ser usados para descifrar el texto cifrado original. La figura 3.3.4 muestra el diagrama de interacción correspondiente a utilizar esta clase para descifrar un determinado texto cifrado en un esquema de umbral.

Es importante notar que *ThresholdDecryptionCombinator* no requiere en ningún momento acceso a las llaves privadas de umbral de ningún miembro de la comisión, solo a t descifrados parciales. *ThresholdDecryptionCombinator* revisa las pruebas de descifrado parcial de cada bloque de descifrado parcial que le es proporcionado mediante *add_partial_decryption*, utilizando como referencia la llave pública de umbral $\overline{k_e}$ y el texto cifrado c dados en el constructor. Esto garantiza que el texto claro recuperado por esta clase sea realmente el mismo que fue cifrado para generar c, usando $\overline{k_e}$ como llave pública. Si no hay suficientes descifrados parciales con pruebas correctas, los métodos *decrypt_to_bitstream* y *decrypt_to_text* fallarán en descifrar c, mas nunca[15] devolverán un descifrado incorrecto. En resumen, *ThresholdDecryptionCombinator* puede ser utilizado por cualquier usuario con acceso

[15] Salvo por la ínfima probabilidad de que algún miembro de la comisión pueda encontrar valores apropiados para falsificar la prueba de Chaum-Pedersen.

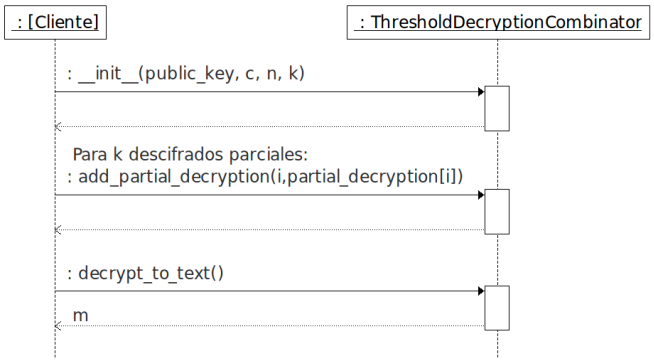

Figura 3.3.4: Diagrama de interacción de clases para la combinación de descifrados parciales

a t presuntos descifrados parciales de $c = Enc\left(\overline{k_e}, m\right)$ para obtener m, sin posibilidad significativa de ser engañado sobre el valor de m.

Podemos observar, de la discusión anterior, que **plonevotecryptolib.Threshold** nos da todas las herramientas necesarias para implementar un esquema de cifrado de umbral con descifrado de umbral distribuido verificable, tal como el que describíamos en la sección 3.1.

Capítulo 4

Redes de mezcla

4.1. Redes de mezcla para la anonimización de votos

4.1.1. Anonimización de votos

El esquema de cifrado de umbral descrito en el capítulo 3 nos proporciona un mecanismo mediante el cual es posible emitir un voto cuyo contenido permanece oculto hasta que un cierto número t de miembros de una comisión elijan revelarlo. Las pruebas de descifrado parcial permiten confiar en que, independientemente de las intenciones de los miembros de la comisión, el valor revelado para cada voto realmente corresponde al voto cifrado por cada elector. Sin embargo, hasta ahora, no hemos visto cómo separar la identidad del elector de su voto. Lo cual es necesario para obtener privacidad en nuestro sistema de votaciones.

Al momento de capturar los votos, independientemente de cómo sea realizado tal proceso de captura[1], es necesario conocer la identidad del elector emitiendo el voto. Esto puesto que es necesario asegurarnos que solo usuarios autorizados voten en una determinada elección y que, además, voten cada uno tan solo un cierto número permitido de veces (generalmente una sola vez por elector). Por otro lado, quisiéramos que cuando los miembros de la comisión descifren el conjunto final de votos capturados, dichos votos no puedan ser relacionados con la identidad del elector que los emitió. Es decir, requerimos contar con un proceso de anonimización de votos entre su captura, y su descifrado y posterior conteo.

Una solución ingenua al problema de anonimización de votos sería pedir al dispositivo de captura de votos (e.g. el servidor de PloneVote) que, tras verificar que el elector está autorizado para votar, guardara el voto por

[1] Veremos los detalles cuando demos el protocolo de votación en la sección 5.2 (particularmente en 5.2.3.2).

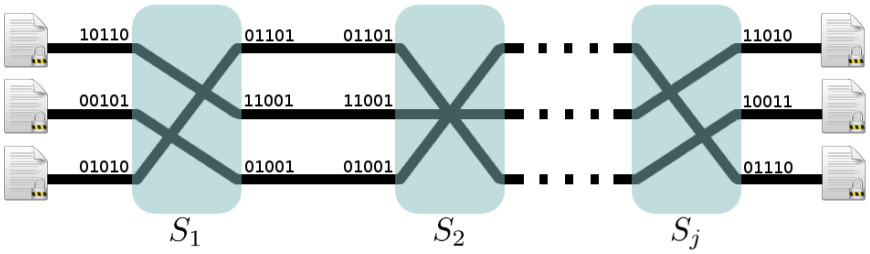

Figura 4.1.1: Esquema general de una red de mezcla.

separado del registro que marca cuáles electores ya votaron, sin asociar tal voto con la identidad del elector. Este esquema coloca toda la responsabilidad sobre la privacidad del sistema en el dispositivo de captura de votos. Un dispositivo de captura malicioso, o funcionando incorrectamente, podría fácilmente filtrar las identidades de los votantes asociadas con sus votos cifrados, o una huella digital de los mismos, a un usuario malicioso. Una vez descifrados los votos al final de la elección, tal usuario podría conocer el valor del voto de cada elector.

Nos gustaría, en cambio, que la privacidad de los votos no dependiera de una sola entidad, sino que múltiples entidades colaboraran para anonimizar los votos, de tal modo que la privacidad del votante solo fuera comprometida si todas estas entidades se encuentran coludidas para atacarla. Un mecanismo que nos permite obtener esta propiedad corresponde a las redes de mezcla.

4.1.2. Esquema general de una red de mezcla

Una red de mezcla es un protocolo en que varias entidades, conocidas como servidores de mezcla, colaboran para procesar un conjunto de mensajes $M = \{m_1, ..., m_l\}$ de varios remitentes, para uno o más destinatarios. Los servidores de mezcla procesan el conjunto[2] M de tal modo que sea imposible para un atacante discernir cuál es el remitente de algún mensaje enviado, aun cuando tal atacante puede observar la comunicación entre todas las entidades involucradas en el protocolo. El esquema general de una red de mezcla es aquel dado en la figura 4.1.1, en que una serie de servidores de mezcla $S_1, ..., S_m$ procesan los votos de la siguiente forma:

1. Cada remitente i cifra su mensaje m_i como c_{i1}, en algún esquema de cifrado que depende del tipo de la red de mezcla, y envía c_{i1} a S_1.

[2]Nuestros conjuntos en este capítulo y el siguiente son realmente arreglos, no conjuntos en el sentido matemático, pues los consideramos ordenados y con la posibilidad de contener elementos duplicados.

2. Una vez que S_j ha recibido todos los mensajes en M cifrados como $C_j = \{c_{1j}, ..., c_{lj}\}$, este servidor aplica una transformación T_j a cada c_{ij}, obteniendo un texto cifrado $c_{i(j+1)} = T_j(c_{ij})$ que contiene el mismo mensaje, pero no puede ser identificado como igual a c_{ij}, salvo tal vez por el legítimo destinatario del mensaje.

3. S_j aplica una permutación aleatoria π_j de los índices en $[1, l]$, y envía el conjunto $C_{j+1} = \left\{c_{\pi_j(1)(j+1)}, ..., c_{\pi_j(l)(j+1)}\right\}$ a S_{j+1}, en el orden dado por tal permutación.

4. S_m envía cada mensaje c_{im} en C_m a su respectivo destinatario[3].

Intuitivamente, cada servidor revuelve o mezcla los mensajes, manteniendo el mismo conjunto M de mensajes (a nivel semántico[4]) pero re-ordenando aleatoriamente sus elementos.

La transformación T_j es usada para evitar que los mensajes del conjunto de salida de un servidor de mezcla puedan ser asociados cada uno con su respectivo mensaje en la entrada del servidor de mezcla, mediante una verificación eficiente de equivalencia. T_j usualmente depende de parámetros secretos conocidos solo por S_j o bien de decisiones aleatorias, que hacen imposible que ninguna entidad distinta de S_j obtenga el mismo valor $c_{i(j+1)} = T_j(c_{ij})$ construido por S_j. Es fácil ver que, si T_j es una función, con todos sus argumentos conocidos por un atacante, dicho atacante puede fácilmente identificar la correspondencia entre la entrada (C_j) del servidor de mezcla y su salida (C_{j+1}), simplemente aplicando T_j a cada elemento de C_j y buscando el resultado entre los elementos permutados en C_{j+1}.

Un atacante que pueda observar tanto la salida como la entrada de un servidor de mezcla, solo podrá identificar qué mensaje en el conjunto de salida corresponde a un determinado mensaje en el conjunto original si conoce la permutación π, o bien el valor de $c_{i(j+1)} = T_j(c_{ij})$ para algún c_{ij}. Un servidor honesto genera π como una permutación aleatoria, eliminándola después de realizado el proceso de mezcla, y jamás la revela a ninguna otra entidad. Un servidor honesto tampoco revela los parámetros secretos de T_j o las decisiones aleatorias tomadas durante su aplicación a un determinado c_{ij}, impidiendo que algún atacante obtenga el $c_{i(j+1)}$ correspondiente.

[3] Este paso es delicado, pues S_m debe tener forma de saber cuál es el destinatario legítimo de c_{im} a partir del mensaje cifrado mismo, o bien enviar todos los mensajes a todos sus destinatarios posibles, y que éstos puedan identificar cuáles son los mensajes que les corresponden (e.g. aquellos que el destinatario puede descifrar). En nuestro escenario esta consideración no es importante, puesto que todos los votos tienen un mismo destinatario: la comisión de la elección.

[4] Cuando decimos que dos textos cifrados son iguales "a nivel semántico" en esta sección, lo que pedimos es que correspondan al mismo mensaje en texto claro m, y además sea claro cuál es el proceso necesario para descifrar cada uno de los dos textos para obtener m (e.g. que el mensaje codificado en cada texto cifrado no sea ambiguo, dependiendo de una elección entre varias formas aparentemente correctas de descifrarlo). Una definición más formal solo puede ser dada para una red de mezcla y esquema de cifrado específicos.

Basta un solo servidor honesto S_j en la red de mezcla, para que C_m sea una permutación aleatoria de M donde ningún mensaje puede ser asociado con su respectivo elemento en C_1, aun por el destinatario legítimo del mensaje[5]. Por lo tanto, una red de mezcla correcta mantendrá el anonimato del remitente de cada mensaje, siempre que al menos un servidor de mezcla siga el protocolo correctamente y no revele T_j o π a ningún atacante.

Es importante notar que el término "servidor de mezcla" se refiere a cualquier entidad que proporcione el servicio de mezclar el conjunto de mensajes, no necesariamente un servidor en el sentido usual de "servidor de red". De hecho, como veremos en la sección 5.2, las acciones de los servidores de mezcla en PloneVote son realizadas por los miembros de la comisión de la elección (y, opcionalmente, auditores independientes), mediante programas cliente ejecutándose en sus equipos de cómputo personales.

4.1.3. Mezclas verificables

Un problema general en redes de mezcla radica en garantizar que cada servidor de mezcla S_j realmente entrega como salida una mezcla de su conjunto de entrada. Esto es, quisiéramos que S_j pudiera probar, a cualquier observador, que el conjunto C_{j+1} realmente contiene los mismos mensajes (a nivel semántico) que el conjunto C_j. Es decir, que al construir C_{j+1}, el servidor de mezcla no eliminó, añadió, reemplazó o modificó ninguno de los mensajes del conjunto original M codificado por C_j.

Si utilizamos en nuestro sistema de votaciones una red de mezcla, donde el proceso de mezcla no es públicamente verificable, entonces un solo servidor de mezcla malicioso puede fácilmente comprometer la correctitud de la elección, añadiendo, eliminando o modificando votos. Intercambiamos así un alto nivel de certeza en la privacidad de la elección, por un muy bajo nivel de certeza en su correctitud. Afortunadamente, no es necesario hacer tal intercambio: podemos construir una red de mezcla públicamente verificable.

En una red de mezcla públicamente verificable, tras mezclar su conjunto de mensajes, cada servidor de mezcla genera y publica una prueba de mezcla que permite a cualquier observador verificar que su conjunto de salida es semánticamente equivalente a su conjunto de entrada (es decir, contiene textos cifrados correspondientes al mismo conjunto de mensajes en texto claro). Por supuesto, quisiéramos que tal prueba no revelara la permutación de los mensajes realizada por S_j, puesto que ello invalidaría cualquier potencial ganancia en privacidad dada por la mezcla así verificada. La figura 4.1.2 ilustra el aparente conflicto entre estos objetivos.

En la sección 4.2.3, veremos, sin embargo, que es posible construir dicha prueba de mezcla. Las pruebas de mezcla que presentaremos son eficientes de generar, eficientes de verificar y no pueden ser falsificadas de forma eficiente más que con probabilidad insignificantemente pequeña.

[5] A no ser, por supuesto, que el texto claro del mensaje contenga información que identifique al remitente.

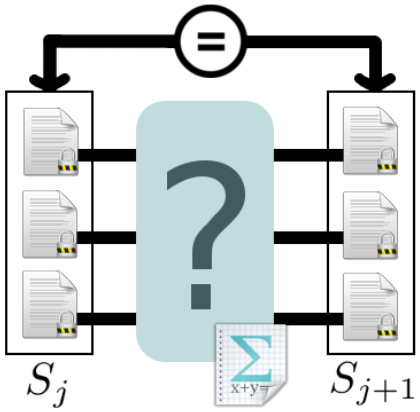

Figura 4.1.2: Objetivos de un proceso de mezcla verificable.

4.2. Teoría: Redes de mezcla mediante recifrado y verificables

4.2.1. Recifrado en ElGamal

Recordemos que en el proceso de cifrado de ElGamal (ver sección 2.2.4), un mismo texto claro m puede ser cifrado, con una misma llave pública k_e, como uno de múltiples textos cifrados distintos. Durante el proceso de cifrado (Alg. 2.4) elegimos un elemento $r \in [1, p-2]$ cualquiera, de forma aleatoria. Para cada una de estas r, el cifrado de m con k_e es

$$c = Enc\,(k_e, m) = (\gamma, \delta) = (g^r \ mod\ p,\ m \cdot g^{ar} \ mod\ p)$$

con $k_d = a$, $k_e = g^a$. Es fácil ver que, para cada posible elección de r, los valores de (γ, δ) resultantes serán distintos.

Un proceso de recifrado en ElGamal queda definido como una transformación *Renc*, tal que dado un texto cifrado $c = Enc\,(k_e, m)$ en una determinada instancia de ElGamal, produce un segundo texto cifrado c' distinto, el cual es también un cifrado de m con k_e en la misma instancia de ElGamal. Quisiéramos, además, que c' tuviera la misma probabilidad de ser cualquiera de los posibles cifrados de m con k_e. Esto con el fin de hacer tan difícil como sea posible la detección de que c' es un recifrado de c.

Es natural esperar que el proceso de recifrado requiera parámetros adicionales al texto cifrado a transformar. Nos gustaría que tales parámetros fueran parámetros públicos de nuestro esquema criptográfico. En el caso de ElGamal, nos gustaría que *Renc* tomara tan solo los parámetros generales de la instancia de ElGamal a ser utilizada (p, g), así como la llave pública k_e.

Observación 21. Si permitimos a *Renc* tomar también la llave privada k_d como parámetro, tenemos una implementación trivial: *Renc* puede simplemente descifrar $m = Dec(k_d, m)$ y cifrarlo de nuevo para obtener $c' = Enc(k_e, m)$, uno de los posibles cifrados de m con k_e tomado con distribución uniforme. Sin embargo, esto requiere que cualquier entidad que realice el proceso de recifrado tenga acceso a la llave privada y, por tanto, sea capaz de leer el mensaje en texto claro m. En una red de mezcla, quisiéramos que varios servidores de mezcla fueran capaces de recifrar un conjunto de textos cifrados, sin tener la capacidad de descifrar su contenido.

Veamos ahora que es posible dar un proceso de recifrado $Renc(k_e, c)$ que toma tan solo el texto cifrado a recifrar y la llave pública k_e con la que fue cifrado, así como los parámetros (p, g) de la instancia de ElGamal utilizada.

Proposición 22. *Dada una instancia de ElGamal (p, g), una llave pública k_e en tal instancia y un texto cifrado $c = Enc(k_e, m)$, es posible, de forma eficiente, obtener c', otro de los posibles cifrados de m con k_e, tomado con distribución uniforme dentro del conjunto de cifrados posibles.*

Demostración. Recordemos que

$$c = Enc(k_e, m) = (\gamma, \delta) = (g^r \bmod p,\ m \cdot g^{ar} \bmod p)$$

para algún $r \in [1, p-2]$. En general, podemos hacer explícita la elección de la r en el proceso de cifrado, y expresar

$$\begin{aligned} c &= c_r \\ &= Enc_r(k_e, m) \\ &= (\gamma, \delta)_r \\ &= (g^r \bmod p,\ m \cdot g^{ar} \bmod p) \\ &= (g^r \bmod p,\ m \cdot (k_e)^r \bmod p). \end{aligned}$$

Ahora, tomemos r' al azar en $\mathbb{Z}^*_{p-1} = \{1, ..., p-2\}$ (con distribución uniforme). Damos entonces

$$\begin{aligned} c' &= (\gamma', \delta') \\ &= \left(\gamma \cdot g^{r'} \bmod p, \delta \cdot (k_e)^{r'} \bmod p\right) \\ &= \left(g^r g^{r'} \bmod p,\ m \cdot (k_e)^r (k_e)^{r'} \bmod p\right) \\ &= \left(g^{r+r'} \bmod p,\ m \cdot (k_e)^{r+r'} \bmod p\right). \end{aligned}$$

Definimos $r'' = r + r' \bmod (p-1)$. Podemos ver que

$$c' = \left(g^{r''} \bmod p,\ m \cdot (k_e)^{r''} \bmod p\right) = Enc_{r''}(k_e, m).$$

Algoritmo 4.1 $Renc\,(p, g, k_e, c)$

1. Generamos un entero r' al azar en $[1, p-2]$
2. Tomamos $(\gamma, \delta) = c$ y generamos $c' = \left(g^{r'}\gamma \ mod\ p, (k_e)^{r'}\ \delta \ mod\ p\right)$
3. Devolvemos c' como el texto recifrado

Al elegir r' con distribución uniforme en $\mathbb{Z}^*_{p-1}$, r'' queda también distribuido uniformemente en ese intervalo[6], independientemente del valor de r. Intuitivamente, podemos pensar que r es solo un "desplazamiento" en $\mathbb{Z}^*_{p-1}$ aplicado a r', y por la estructura cíclica de tal conjunto, si r' está uniformemente distribuido en el conjunto, r'' también lo estará. Por lo tanto, c' está tomado con distribución uniforme dentro del conjunto de posibles cifrados $Enc\,(k_e, m)$ de m con k_e. □

El algoritmo 4.1 muestra el proceso de recifrado en ElGamal dado por la proposición anterior. Cuando sea conveniente para nuestra explicación, obviaremos los parámetros p y g, dando el recifrado como $Renc\,(k_e, c)$.

Dados c y c', dos textos cifrados en un esquema ElGamal, junto con los parámetros públicos de la instancia de ElGamal en uso y la llave pública k_e, no se conoce forma eficiente de determinar si c' es un recifrado de c. Es decir, si $c = Enc_r\,(k_e, m)$ y $c' = Enc_{r''}\,(k_e, m)$ para dos $r', r'' \in [1, p-2]$ y mismos k_e y m. Lo cual nos permite usar el recifrado como nuestra transformación (T_j) para anonimizar mensajes dentro de una red de mezcla.

4.2.1.1. Pruebas de recifrado

De forma análoga a como hicimos para Enc_r, denotemos por $Renc_{r'}\,(p, g, k_e, c)$ al recifrado en ElGamal con $r' \in [1, p-2]$ elegido como el parámetro aleatorio.

Es importante notar que, dada la información adicional del r' elegido durante el proceso de recifrado, es trivial verificar que c' es un recifrado de c usando r', como lo muestra la siguiente proposición.

Proposición 23. *Dados c y c' dos textos cifrados en una misma instancia (p, g) de ElGamal, así como la llave pública k_e con la que fue cifrado c y un valor r', es posible verificar, de forma eficiente, si c' es un recifrado de c con parámetro aleatorio r'.*

[6] Existe el caso patológico en que $r + r'\ mod\ (p-1) = 0$. Podemos detectar este caso, pues cuando esto ocurre, $c' = (1, m)$, y podemos corregirlo volviendo a elegir un nuevo r' al azar. La probabilidad de encontrar este caso es $1/|p-1|$ y nos da el descifrado de c (es equivalente a obtener m de c_r adivinando r al azar). Notemos que podemos aplicar $Renc_{r'}$ a tal c' y obtendremos simplemente m cifrado con r' (e.g. $Enc_{r'}\,(k_e, m)$).

Demostración. Fijado un r' especifico,

$$Renc_{r'}(p, g, k_e, (\gamma, \delta)) = \left(g^{r'}\gamma \bmod p, (k_e)^{r'}\delta \bmod p\right).$$

Es fácil ver que $Renc_{r'}$ es una función en el sentido matemático. Esto es, su salida está unívocamente determinada por su entrada. Para ver si $c' = Renc_{r'}(p, g, k_e, c)$, basta entonces construir $c'' = Renc_{r'}(p, g, k_e, c)$ y verificar si c'' y c' son iguales. Si lo son, entonces c' es un recifrado de c con parámetro aleatorio r'. En caso contrario c' no es un recifrado de c con parámetro aleatorio r'.

La eficiencia de esta verificación es consecuencia inmediata de la eficiencia de *Renc*. □

Una segunda propiedad de recifrado en ElGamal que nos será útil más adelante es la capacidad de mostrar, de forma eficiente, que dos textos cifrados $c_1 = Renc_{r_1}(p, g, k_e, c)$ y $c_2 = Renc_{r_2}(p, g, k_e, c)$, generados como recifrados de un mismo c, cifran realmente el mismo mensaje m. Quisiéramos, además, poder mostrar lo anterior sin revelar que c_1 y c_2 son recifrados de c.

Podríamos mostrar que c_1 y c_2 cifran el mismo mensaje simplemente revelando c, r_1 y r_2, lo que permitiría a cualquier verificador usar el método dado por la proposición 23 para ver que c_1 es un recifrado de c lo mismo que c_2, y por tanto c_1 y c_2 codifican un mismo mensaje m con llave pública k_e. El problema radica en probar la equivalencia entre c_1 y c_2 sin revelar la equivalencia de ninguno de los dos con c.

Para mostrar dicha equivalencia, nos gustaría, en su lugar, poder expresar c_2 en cambio como un recifrado directo de c_1, sin involucrar a c en el proceso. Si esto fuera posible, podemos revelar el parámetro aleatorio $\tilde{r}$ de tal recifrado para mostrar la equivalencia entre c_1 y c_2, sin revelar información adicional sobre c.

Proposición 24. *Dados $c_1 = Renc_{r_1}(p, g, k_e, c)$ y $c_2 = Renc_{r_2}(p, g, k_e, c)$, dos recifrados del mismo $c = Enc_r(k_e, m)$ en una cierta instancia (p, g) de ElGamal, podemos mostrar, de forma eficiente, que c_1 y c_2 cifran el mismo mensaje m, con llave pública k_e, sin revelar su conexión con c.*

Demostración. Notemos que

$$\begin{aligned}
c_2 &= \left(g^{r+r_2} \bmod p,\ m \cdot (k_e)^{r+r_2} \bmod p\right) \\
&= \left(g^{r+r_1-r_1+r_2} \bmod p,\ m \cdot (k_e)^{r+r_1-r_1+r_2} \bmod p\right) \\
&= \left(g^{(r+r_1)+(r_2-r_1)} \bmod p,\ m \cdot (k_e)^{(r+r_1)+(r_2-r_1)} \bmod p\right) \\
&= \left(g^{(r+r_1)+\tilde{r}} \bmod p,\ m \cdot (k_e)^{(r+r_1)+\tilde{r}} \bmod p\right) \\
&= Renc_{\tilde{r}}(p, g, k_e, c_1)
\end{aligned}$$

con $\tilde{r} = r_2 - r_1 \ mod \ (p-1)$. Revelando tan solo $\tilde{r}$, podemos probar que c_1 y c_2 codifican el mismo mensaje m con llave pública k_e.

Calcular $\tilde{r}$ puede ser realizado de forma eficiente conociendo r_1 y r_2. Una vez generado y publicado $\tilde{r}$, la verificación de la equivalencia entre c_1 y c_2 se lleva a cabo de la misma forma que en la proposición 23, y por tanto puede ser realizada eficientemente.

Conocer $\tilde{r}$ no permite conocer r_1 o r_2, a no ser que alguno de los dos ya sea conocido previamente, pues para cada $\tilde{r}$ y r_1 dados, existe un r_2 tal que $\tilde{r} = r_2 - r_1 \ mod \ (p-1)$, y análogamente para $\tilde{r}$ y r_2 dados. Por lo tanto, $\tilde{r}$ no da información que permita verificar que c_1 o c_2 son recifrados de c, a no ser que c ya haya sido mostrado como equivalente a alguno de los dos c_i. □

En la sección 4.2.3, veremos cómo usar los dos tipos de pruebas anteriores para dar una prueba de mezcla con las propiedades descritas en la sección 4.1.3.

4.2.2. Construcción de una red de mezcla

Existen dos tipos principales de redes de mezcla en la literatura: redes de mezcla por descifrado y redes de mezcla por recifrado.

En PloneVote utilizaremos exclusivamente redes de mezcla por recifrado. Sin embargo, por completitud y perspectiva histórica, consideramos apropiado describir también el funcionamiento de redes de mezcla por descifrado, mencionando brevemente las razones por las cuales no utilizamos tales redes de mezcla en nuestro sistema de votaciones.

4.2.2.1. Redes de mezcla por descifrado

La primera construcción para una red de mezcla por descifrado, así como la introducción del concepto de red de mezcla, se deben a Chaum [Chaum1981].

Chaum imaginaba dicha red siendo usada para intercambiar correo electrónico de forma que el remitente permaneciera anónimo. Consiguientemente, describe un mecanismo para codificar el destinatario de cada mensaje en el mensaje cifrado c_{i1} generado por el remitente. Esta dirección codificada permite que el mensaje pueda ser direccionado a su destinatario correcto sin revelar su origen, ni siquiera a dicho destinatario o al último servidor de mezcla en la red. Daremos a continuación el caso simplificado de la red de mezcla de Chaum con un solo destinatario.

En la red de mezcla de Chaum, cada servidor de mezcla S_j tiene su propio par de llaves (k_{ej}, k_{dj}) en algún esquema de cifrado de llave pública. Del mismo modo, el destinatario D tiene su propio par de llaves (k_e, k_d). Las llaves públicas k_e y k_{ej} (para cada j), son conocidas por todos los remitentes.

Para enviar un mensaje m_i, un remitente genera

$$c_{i1} = Enc\left(k_{e1}, Enc\left(k_{e2}, ...Enc\left(k_{en}, Enc\left(k_e, m_i\right)\right)...\right)\right),$$

y envía tal mensaje a S_1. Cuando S_j recibe el conjunto de mensajes, éste descifra cada mensaje con su propia llave privada (es decir, $T_j(x) = Dec(k_{dj}, x)$), permuta aleatoriamente el conjunto, y pasa el resultado al siguiente servidor S_{j+1}. El último servidor de mezcla, S_n, transmite su salida a D, el cual descifra cada mensaje con su llave privada, obteniendo los mensajes del conjunto $M = \{m_1, ..., m_l\}$ en orden aleatorio.

Intuitivamente, este esquema funciona cifrando el mensaje original por capas, una capa por cada servidor de mezcla y el destinatario final, aplicadas en orden inverso. Cada servidor en la red de mezcla retira la capa de cifrado que le corresponde, permuta los mensajes y los envía al siguiente servidor en la red.

Una consecuencia importante de lo anterior es que la lista de servidores de mezcla queda decidida en el momento en que es cifrado cada mensaje y, a partir de ese punto, el mensaje deberá pasar, en orden, por todos esos servidores antes de poder ser leído por el destinatario. No existe forma de saltarse un servidor de mezcla, por lo que cualquiera de los servidores puede detener la transmisión del conjunto de mensajes. Por otro lado, si el destinatario es un posible atacante, éste no puede comprometer el anonimato de los mensajes, aun si pudiera verlos antes de que pasaran por el primer servidor en la red de mezcla.

PloneVote no utiliza una red de mezcla por descifrado, en parte por la fragilidad que resulta de que un solo servidor de mezcla puede parar la transmisión de todos los mensajes, en parte porque no contamos con (o no conocemos) técnicas para garantizar la correctitud del proceso de mezcla realizado por cada servidor de mezcla en dicho esquema[7]. En vez de eso, nuestro protocolo para elecciones hace uso de una red de mezcla por recifrado con mezcla verificable.

4.2.2.2. Redes de mezcla por recifrado

Una alternativa a las redes de mezcla por descifrado son las redes de mezcla por recifrado. El primer ejemplo de un esquema de red de mezcla por recifrado está dado por [PIK1994], y es la base del esquema implementado en PloneVoteCryptoLib. Dicha red de mezcla hace uso de la propiedad de recifrado de ElGamal[8] vista en la sección 4.2.1, que permite transformar un texto cifrado $c = Enc(k_e, m)$ en otro texto cifrado $c' = Renc(k_e, c)$, codificando el mismo mensaje m, sin necesidad de conocer la llave privada k_d o poder descifrar c.

[7] Notemos que un servidor de mezcla S_j puede modificar de forma silenciosa cualquier mensaje en el conjunto, simplemente entregando $Enc(k_{e(j+1)}, ...Enc(k_e, m_{i'})...)$ a S_{j+1}, en lugar de $Dec(k_{dj}, c_{ij})$.

[8] Varios otros esquemas de cifrado de llave pública presentan propiedades de recifrado similares, permitiendo también implementar redes de mezcla análogas sobre tales esquemas. Sin embargo, ElGamal es el esquema más comúnmente utilizado en la práctica que soporta recifrado.

A continuación describiremos el funcionamiento de esta red de mezcla. Consideraremos la instancia (p, g) de ElGamal como fija para todas nuestras operaciones criptográficas.

El cifrado original de los mensajes en M es realizado como $c_{i1} = Enc(k_e, m)$, en una sola capa, con la llave pública del destinatario final del mensaje. Cada servidor de mezcla S_j recibe el conjunto de mensajes cifrados $C_j = \{c_{1j}, ..., c_{lj}\}$ y transforma cada c_{ij} en $c_{i(j+1)} = Renc(k_e, c_{ij})$, otro de los posibles cifrados del mensaje original cifrado por c_{ij}. Esto es, $T_j(x) = Renc(k_e, x)$ para toda j, donde el parámetro aleatorio r' del recifrado es elegido independientemente para cada c_{ij} que S_j recibe. Finalmente, S_j genera una permutación aleatoria π de los índices en $[1, l]$, construye el conjunto $C_{j+1} = \left\{c_{\pi_j(1)(j+1)}, ..., c_{\pi_j(l)(j+1)}\right\}$ y lo transmite al siguiente servidor de mezcla.

Llamamos a esta composición de recifrado y permutación aleatoria realizada por cada servidor de mezcla, el proceso de mezcla. Como en general no hay forma eficiente de verificar si c' es un recifrado de c o no, sin conocer el parámetro aleatorio r' del recifrado, el proceso de mezcla genera un conjunto con los mismos mensajes que C_j, cifrados con la misma llave pública, pero imposibles de asociar uno a uno con los mensajes de C_j.

En contraste con la rigidez de las redes de mezcla por descifrado, en una red de mezcla por recifrado cualquier usuario con acceso a un conjunto de mensajes cifrados C_j puede realizar el proceso de mezcla. El destinatario final de los mensajes puede usar k_d, la llave privada correspondiente a la llave pública k_e, para descifrar los mensajes dentro de cualquier C_j. De esta forma, los servidores de mezcla involucrados en el protocolo no quedan fijos al momento de cifrar los mensajes, cualquier usuario puede ser añadido como un servidor de la red de mezcla y ningún servidor de mezcla puede detener el proceso, ya que es posible saltarse un proceso de mezcla sin perder más que la posible privacidad proporcionada por tal servidor.

Una desventaja de lo anterior es que, si la identidad de los remitentes para los mensajes en M debe ser protegida también del destinatario, entonces debemos contar con algún método para asegurar que han sido realizadas suficientes mezclas sobre estos mensajes (por entidades que podemos suponer no se encuentran coludidas), antes de que el destinatario descifre el conjunto de textos cifrados. Por ejemplo, si el destinatario puede obtener acceso a C_1 y descifrarlo, entonces claramente puede conocer la identidad del remitente asociado a cada mensaje de M, puesto que C_1 no ha sido permutado aún. Veremos las implicaciones de esto para nuestro protocolo de votaciones en la sección 5.2.

4.2.3. Prueba de mezcla en conocimiento cero

Tal como está dada nuestra red de mezcla en la sección 4.2.2.2, nada impide a un servidor de mezcla malicioso, S_j, reemplazar un mensaje c_{ij} por un mensaje m' de su elección, generando $c_{i(j+1)} = Enc(k_e, m')$ en vez

de $c_{i(j+1)} = Renc(k_e, c_{ij})$. Nos gustaría, para evitar tal posibilidad, que S_j pudiera probar que su conjunto de salida C_{j+1} contiene exactamente los mismos mensajes a nivel semántico que C_j. En este punto podemos proveer una definición precisa de "a nivel semántico".

Definición 25. Equivalencia semántica de conjuntos de textos cifrados: Dada una instancia de ElGamal y una llave pública k_e en tal instancia, decimos que dos conjuntos C y C' de textos cifrados son *semánticamente equivalentes* (para tal instancia y llave), denotado $C \approx_S C'$, si para cada mensaje m posible, existen el mismo número de textos cifrados $c \in C$ tales que $c = Enc_r(k_e, m)$ para algún r, que de textos cifrados $c' \in C'$ tales que $c' = Enc_{r'}(k_e, m)$ para algún r'.

Supongamos que S_j recifra cada c_{ij} con parámetro aleatorio r_i. Es decir, $c_{i(j+1)} = Renc_{r_i}(k_e, c_{ij})$. Además, denotemos por π_j a la permutación de los índices realizada por S_j.

Trivialmente, S_j podría probar que el C_{j+1} que produce mediante el proceso de mezcla es equivalente semánticamente al conjunto C_j de su entrada, simplemente publicando $r_1, ..., r_l$ y π_j. Sin embargo, esto permitiría asociar cada elemento de C_j con su correspondiente elemento de C_{j+1}, eliminando la contribución a la privacidad dada por el proceso de mezcla.

Como mencionamos en la sección 4.1.3, lo que realmente quisiéramos es que S_j pudiera producir una prueba públicamente verificable de que C_{j+1} es un conjunto de textos cifrados semánticamente equivalente a C_j, sin revelar la relación de ningún elemento dentro de C_{j+1} con algún elemento de C_j. El lector astuto notará que esto es equivalente a decir que buscamos una prueba en conocimiento cero de $C_j \approx_S C_{j+1}$ (ver sección 3.2.3.1). Una forma de construir dicha prueba es la descrita a continuación, dada por Benaloh en [Benaloh2006].

Se elige a priori un parámetro de seguridad α (e.g. $\alpha = 128$). Para probar la equivalencia semántica de C_j y C_{j+1}, S_j genera primero una serie de mezclas $\overline{C_{j1}}, ..., \overline{C_{j\alpha}}$ de C_j, distintas entre sí y distintas de C_{j+1}, siguiendo el mismo proceso de mezcla que usó para generar C_{j+1}. Para cada $\overline{C_{j\beta}}$, S_j conserva, de momento, los parámetros $r_{\beta 1}, ..., r_{\beta l}$ y $\pi_{j\beta}$, tales que $\overline{C_{j\beta}} = \left\{c_{\pi_{j\beta}(1)j\beta}, ..., c_{\pi_{j\beta}(l)j\beta}\right\}$, y $c_{ij\beta} = Renc_{r_{\beta i}}(k_e, c_{ij})$, así como los parámetros $r_1, ..., r_l$ y π_j, usados para generar C_{j+1}.

En la versión interactiva de la prueba en conocimiento cero, S_j envía primero C_{j+1} y $\overline{C_{j1}}, ..., \overline{C_{j\alpha}}$ a un verificador honesto V. Suponemos que V ya conoce C_j. V responde a S_j con un reto φ, formado por una secuencia de α bits aleatorios. Para cada $\beta \in [1, \alpha]$, sea b el bit en la posición β de φ:

- Si $b = 0$, S_j muestra a V la equivalencia entre C_j y $\overline{C_{j\beta}}$, revelando los valores de los parámetros $r_{\beta 1}, ..., r_{\beta l}$ y $\pi_{j\beta}$ (ver proposición 23).

- Si $b = 1$, S_j muestra a V la equivalencia entre $\overline{C_{j\beta}}$ y C_{j+1}, calculando $\tilde{r}_{\beta i} = r_i - r_{\beta i} \ mod \ (p-1)$ para cada i y revelando los valores de los

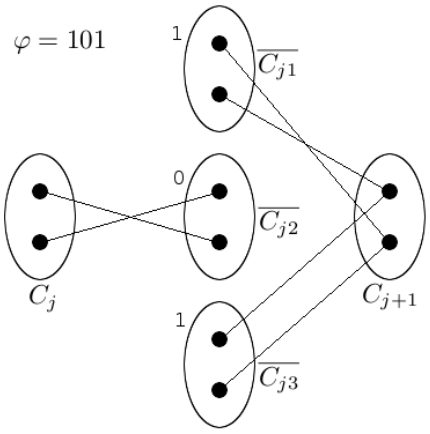

Figura 4.2.1: Ejemplo de asociaciones reveladas en nuestra prueba en conocimiento cero de $C_j \approx_S C_{j+1}$.

parámetros $\tilde{r}_{\beta 1}, ..., \tilde{r}_{\beta l}$ (ver proposición 24), así como la permutación $\pi_j \circ \pi_{j\beta}^{-1}$ entre los elementos de $\overline{C_{j\beta}}$ y los de C_{j+1}.

La figura 4.2.1 muestra las asociaciones reveladas por S_j en un escenario particular en que $l = 2$, $\alpha = 3$ y $\varphi = 101$. Claramente estos parámetros son artificialmente pequeños, pero nos permiten visualizar la información dada a V por S_j para mostrar $C_j \approx_S C_{j+1}$. Notemos, en particular, que ningún elemento de C_j es asociado con algún elemento de C_{j+1} directa o indirectamente (e.g. a través de los elementos de alguno de los conjuntos $\overline{C_{j\beta}}$).

La prueba funciona, pues S_j genera primero los conjuntos $\overline{C_{j1}}, ..., \overline{C_{j\alpha}}$ y, para cada $\overline{C_{j\beta}}$, con igual probabilidad, deberá poder probar que dicho conjunto es equivalente a C_j o a C_{j+1}. Si $\overline{C_{j\beta}}$ no es equivalente a ambos conjuntos, entonces S_j podrá responder correctamente al β-ésimo bit del reto φ de V solo la mitad de las veces. Si C_j y C_{j+1} no son equivalentes, ningún $\overline{C_{j\beta}}$ generado por S_j podrá ser equivalente a ambos conjuntos, y la probabilidad de que S_j pueda responder correctamente al reto φ sera $1/2^\alpha$. Claramente, para α suficientemente grande (e.g. ≥ 100), tal probabilidad es despreciable. Por lo tanto, si S_j puede responder al reto, tendremos un alto nivel de confianza en que, en efecto, $C_j \approx_S C_{j+1}$.

Además, la prueba es en conocimiento cero, pues S_j nunca revela la relación entre los elementos de C_j y C_{j+1} de forma explícita o implícita. Si S_j revelara la relación entre los elementos de C_j y $\overline{C_{j\beta}}$, y la relación entre los elementos de $\overline{C_{j\beta}}$ y C_{j+1} para un mismo β, entonces sería trivial reconstruir la relación entre los elementos de C_j y C_{j+1}. Sin embargo, para cada β, S_j solo prueba una de las dos relaciones.

Una manera algo más formal de mostrar que la prueba anterior es de co-

nocimiento cero es mediante el argumento de simulación. Es trivial construir un simulador de la interacción entre S_j y V que "muestre" que C_j y C_{j+1} son semánticamente equivalentes, aun cuando éstos no lo sean. Basta invertir el orden en que φ y $\overline{C_{j1}}, ..., \overline{C_{j\alpha}}$ son generados. Si podemos generar los conjuntos $\overline{C_{j\beta}}$ tras conocer el reto φ, entonces podemos simular la prueba de equivalencia entre cualquier par de conjuntos de textos cifrados con el mismo número de elementos, construyendo $\overline{C_{j\beta}}$ como una mezcla de C_j cuando el β-ésimo bit de φ es 0, y como una mezcla de C_{j+1} cuando dicho bit es 1. Es claro, entonces, que la información provista por la prueba de $C_j \approx_S C_{j+1}$ no puede contener la información de la relación entre los elementos de C_j y los elementos de C_{j+1}.

Tal como hicimos en la sección 3.2.3 para la prueba en conocimiento cero de descifrado parcial, podemos usar la heurística de Fiat-Shamir para eliminar la necesidad de contar con un verificador V en nuestra prueba de equivalencia semántica de C_j y C_{j+1}. En vez de recurrir a V para generar el reto φ, lo construimos a partir de una función hash aplicada al conjunto $\left\{C_j, C_{j+1}, \overline{C_{j1}}, ..., \overline{C_{j\alpha}}\right\}$. Esto tiene la ventaja adicional de generar una prueba públicamente verificable, lo cual no ocurre en el caso de la prueba interactiva.

Si la función de hash no es predecible, construir $\overline{C_{j1}}, ..., \overline{C_{j\alpha}}$ adecuados para falsificar la prueba de equivalencia semántica de C_j y C_{j+1} solo podrá conseguirse con probabilidad $1/2^{\alpha}$. Esto requiere, en promedio, $2^{\alpha-1}$ intentos. Para α suficientemente grande, generar tales conjuntos no es factible en la práctica en ningún tiempo razonable.

El algoritmo 4.2 muestra el proceso de mezcla completo realizado por cada servidor de mezcla en nuestra red de mezcla por recifrado verificable, tal como es implementado en PloneVoteCryptoLib. En el algoritmo, p es el primo usado por la instancia de ElGamal, k_e la llave pública en tal instancia usada para cifrar cada mensaje y H es una función de hash segura que genera al menos α bits de salida.

La prueba $\wp$ puede ser verificada posteriormente por cualquier usuario, simplemente volviendo a obtener el hash de $\left\{C_j, C_{j+1}, \overline{C_{j1}}, ..., \overline{C_{j\alpha}}\right\}$ y revisando cada prueba de equivalencia entre $\overline{C_{j\beta}}$ y C_j o C_{j+1}, realizando los recifrados apropiados con los parámetros aleatorios revelados, seguidos de la permutación correspondiente y verificando que el resultado sea el esperado. El algoritmo 4.3 muestra el proceso de verificación de una prueba de equivalencia semántica entre C_j y C_{j+1}, realizado, a posteriori, por un agente cualquiera A. Usamos la notación $C\,[i]$ para referirnos al elemento i del conjunto (ordenado) de textos cifrados C.

Algoritmo 4.2 Proceso de mezcla con generación de prueba públicamente verificable.

1. S_j recupera el conjunto $C_j = \{c_{1j}, ..., c_{lj}\}$ de textos cifrados a mezclar
2. Para $i \in [1, l]$:
   a) S_j elige $r_i \in [1, p-2]$ al azar y calcula $c_{i(j+1)} = Renc_{r_i}(k_e, c_{ij})$
3. S_j genera una permutación aleatoria π_j de los indices en $[1, l]$
4. S_j construye $C_{j+1} = \left\{c_{\pi_j(1)(j+1)}, ..., c_{\pi_j(l)(j+1)}\right\}$
5. Para $\beta \in [1, \alpha]$:
   a) Para $i \in [1, l]$:
      1) S_j elige $r_{\beta i} \in [1, p-2]$ al azar y calcula $c_{ij\beta} = Renc_{r_{\beta i}}(k_e, c_{ij})$
   b) S_j genera una permutación aleatoria $\pi_{j\beta}$ de los índices en $[1, l]$
   c) S_j construye $\overline{C_{j\beta}} = \left\{c_{\pi_{j\beta}(1)j\beta}, ..., c_{\pi_{j\beta}(l)j\beta}\right\}$
6. S_j calcula $\varphi = H\left(C_j, C_{j+1}, \overline{C_{j1}}, ..., \overline{C_{j\alpha}}\right)$
7. S_j construye una prueba vacía $\wp$
8. Para $\beta \in [1, \alpha]$:
   a) S_j obtiene el β-ésimo bit b de φ
   b) Si $b = 0$:
      1) S_j añade el bloque $\wp[\beta] = \left((r_{\beta 1}, ..., r_{\beta l}), \pi_{j\beta}\right)$ a $\wp$
   c) Si $b = 1$:
      1) Para cada $i \in [1, l]$: S_j calcula $\tilde{r}_{\beta i} = r_i - r_{\beta i} \ mod \ (p-1)$
      2) S_j calcula $\tilde{\pi}_{j\beta} = \pi_j \circ \pi_{j\beta}^{-1}$
      3) S_j añade el bloque $\wp[\beta] = \left((\tilde{r}_{\beta 1}, ..., \tilde{r}_{\beta l}), \tilde{\pi}_{j\beta}\right)$ a $\wp$
9. S_j publica C_{j+1}, $\overline{C_{j1}}, ..., \overline{C_{j\alpha}}$ y $\wp$

Algoritmo 4.3 Verificación de una prueba de mezcla por un agente cualquiera A.

1. A obtiene C_j, C_{j+1}, $\overline{C_{j1}}, ..., \overline{C_{j\alpha}}$ y la prueba de mezcla $\wp$
2. A calcula $\varphi = H\left(C_j, C_{j+1}, \overline{C_{j1}}, ..., \overline{C_{j\alpha}}\right)$
3. Para $\beta \in [1, \alpha]$:
   a) A obtiene el β-ésimo bit b de φ
   b) A recupera el β-ésimo bloque de $\wp$ como $\left(\left(r_1, ..., r_l\right), \pi\right) = \wp\left[\beta\right]$
   c) Si $b = 0$:
      1) Para $i \in [1, l]$: A calcula $c'_i = Renc_{r_i}\left(k_e, C_j\left[i\right]\right)$
      2) A calcula $C' = \left\{c'_{\pi(1)}, ..., c'_{\pi(l)}\right\}$
      3) A verifica que $\overline{C_{j\beta}} = C'$, rechazando la prueba en caso contrario
   d) Si $b = 1$:
      1) Para $i \in [1, l]$: A calcula $c'_i = Renc_{r_i}\left(k_e, \overline{C_{j\beta}}\left[i\right]\right)$
      2) A calcula $C' = \left\{c'_{\pi(1)}, ..., c'_{\pi(l)}\right\}$
      3) A verifica que $C_{j+1} = C'$, rechazando la prueba en caso contrario

4.3. Implementación: Redes de mezcla en PloneVoteCryptoLib

4.3.1. Descripción y uso

El paquete **plonevotecryptolib.Mixnet**, dentro de PloneVoteCryptoLib, concentra las clases relacionadas con la implementación de una red de mezcla públicamente verificable para su uso en elecciones en PloneVote.

En comparación con la implementación de cifrado de umbral, la interfaz externa de este paquete es considerablemente más sencilla. El cliente de PloneVoteCryptoLib requiere conocer solo dos clases de este paquete para realizar o verificar procesos de mezcla de votos: *CiphertextCollection* y *ShufflingProof*. A continuación describimos el uso esperado de estas clases dentro de PloneVote[9]:

1. Una vez capturados todos los votos en una elección, construimos un nuevo objeto *CiphertextCollection*, y añadimos a éste uno por uno

[9] El contexto de este uso y los agentes que realizan cada acción serán detallados en la sección 5.2.3.

los votos, mediante el método *add_ciphertext*. Este objeto representa nuestro conjunto inicial de entrada a la red de mezcla C_1 (ver sección 4.1.2).

2. Cada servidor de mezcla S_j recibe su conjunto de votos cifrados C_j como un objeto *CiphertextCollection*. Para realizar el proceso de mezcla verificable, S_j solo debe llamar al método *shuffle_with_proof* de esta clase, el cual devolverá el conjunto mezclado C_{j+1} (como otro objeto de clase *CiphertextCollection*), junto con la prueba en conocimiento cero de que $C_j \approx_S C_{j+1}$, codificada como un objeto de clase *ShufflingProof*.

3. Cualquier otro usuario con acceso a C_j, C_{j+1} y su respectiva prueba de equivalencia semántica, puede verificar tal prueba utilizando el método *verify* de *ShufflingProof*.

Por supuesto, tanto *CiphertextCollection* como *ShufflingProof* pueden ser guardados en (y cargados desde) archivos, para facilitar su transferencia entre usuarios, publicación y almacenamiento.

Mientras los servidores de mezcla, así como los auditores que deseen verificar el proceso de mezcla, estén usando una versión de PloneVoteCryptoLib en la cual confíen[10], éstos no deben preocuparse por los detalles de la generación y verificación de las pruebas de mezcla en conocimiento cero dadas por los algoritmos 4.2 y 4.3. Tampoco necesitan preocuparse porque el proceso de mezcla anonimice correctamente los votos, ya que *shuffle_with_proof* genera la permutación y los parámetros aleatorios de recifrado usando el generador de números aleatorios de pycrypto [pycrypto], considerado seguro, y olvida tal información antes de devolver sus resultados al cliente. Por supuesto, si algún auditor no confía en ninguna versión disponible de PloneVoteCryptoLib, las pruebas de mezcla guardadas como archivos XML proporcionan suficiente información para verificar la validez del proceso de mezcla correspondiente, ya sea a mano o mediante otro programa.

La sección 4.3.2 explora a más detalle estas dos clases y sus métodos, así como el resto de las clases de **plonevotecryptolib.Mixnet**, utilizadas internamente para soportar el proceso de mezcla o su verificación.

4.3.1.1. Limitaciones e indicaciones adicionales

Un detalle importante a notar, es que nuestro proceso de mezcla solo será efectivo en anonimizar los votos dentro de una colección de votos cuando tales votos tengan todos la misma longitud en bloques.

Recordemos que PloneVoteCryptoLib cifra cada mensaje o voto como una secuencia de bloques $c = (\gamma, \delta)$. El proceso de mezcla realizado por *shuffle_with_proof* recifra cada texto cifrado por bloques, devolviendo un texto cifrado con el mismo número de bloques que el texto original, pero

[10] Por ejemplo, una versión cuya huella digital corresponde a la publicada por varios auditores independientes que garantizan la correctitud del código.

donde cada bloque ha sido recifrado individualmente siguiendo el algoritmo 4.1. Sin contar con la información específica del recifrado, es imposible asociar un bloque recifrado al bloque de texto cifrado original desde el cual fue obtenido. Sin embargo, los textos cifrados en el conjunto mezclado tendrán cada uno la misma longitud en bloques que el texto cifrado correspondiente del conjunto original. Si los textos cifrados del conjunto original tienen longitudes diferentes (en bloques) esto debilita la privacidad obtenida por el proceso de mezcla[11].

Al momento de escribir este documento, la solución soportada por PloneVoteCryptoLib para mitigar este problema, es utilizar el parámetro *pad_to* de *PublicKey.encrypt_X* para forzar un tamaño mínimo en todos los votos cifrados de una misma elección (ver sección 2.3.2). Dicho tamaño debe ser elegido cuidadosamente, de tal modo que sea el tamaño más pequeño que garantice que ningún voto válido en la elección exceda tal tamaño. Un tamaño demasiado pequeño causará que algunos votos excedan ese tamaño y sean identificables por ello en el proceso de mezcla. Por otro lado, un tamaño excesivo aumenta correspondientemente los costos de almacenar y procesar los votos durante la elección. Afortunadamente, esperamos que, en la mayoría de las elecciones, el esquema de votación usado dicte un tamaño máximo acotado de votos y relativamente pequeño (probablemente de un solo bloque en la mayoría de los casos).

Trabajo futuro podría explorar métodos viables para adjuntar bloques aleatorios a objetos *Ciphertext* ya construidos, sin alterar el mensaje codificado en éstos, sin requerir la llave privada y de forma públicamente verificable.

4.3.2. Arquitectura

La figura 4.3.1 muestra las clases principales dentro de PloneVoteCryptoLib involucradas en dar soporte a mezclas verificables y redes de mezcla por recifrado. Al igual que en la sección 2.3.2, omitimos alguna información no esencial de nuestro esquema, en particular aquella referente a las excepciones que pueden ser lanzadas por algunos métodos cuando sus parámetros son incorrectos o fallan las verificaciones de seguridad. Salvo por la interfaz *Storable*, marcada en gris y descrita en la sección 2.3.2, todas estas clases forman parte del paquete **plonevotecryptolib.Mixnet**.

A continuación describimos en orden cada una de las clases en la figura 4.3.1, junto con sus métodos y atributos públicos. Como veremos, a pesar de la simplicidad de su interfaz externa, el funcionamiento interno de **plonevotecryptolib.Mixnet** es considerablemente complejo.

[11] En la práctica, es como si separáramos el conjunto original en subconjuntos de textos cifrados con la misma longitud en bloques y mezcláramos cada subconjunto por separado. Los textos cifrados solo son anónimos dentro del subconjunto de textos cifrados con el mismo número de bloques.

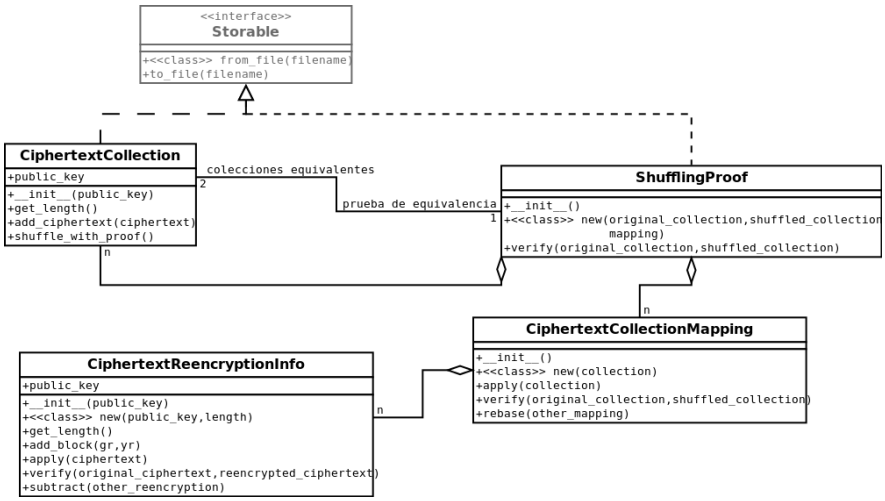

Figura 4.3.1: Diagrama de clases simplificado para soporte de mezclas verificables por recifrado en ElGamal en PloneVoteCryptoLib

CiphertextCollection

CiphertextCollection representa una colección (un conjunto ordenado con posibilidad de repetición) de textos cifrados con la misma llave pública en una misma instancia del esquema ElGamal. El atributo *public_key* contiene la llave pública común de todos los textos cifrados en la colección. Dicho valor es, también, el único parámetro tomado por el constructor de *CiphertextCollection*.

El método *CiphertextCollection.add_ciphertext* permite añadir un texto cifrado a la colección. Dicho método verifica que el texto cifrado dado haya sido generado con la llave pública correcta, antes de aceptarlo.

CiphertextCollection es iterable e indizable como colección de objetos de tipo *Ciphertext*, implementando los métodos especiales de python *__iter__* y *__getitem__*. Adicionalmente, *CiphertextCollection* implementa *__eq__* y *__neq__*, permitiendo verificar igualdad (==, !=) entre dos colecciones de textos cifrados, con el significado natural: dos colecciones son iguales solo si sus elementos son iguales uno a uno y en el mismo orden. *CiphertextCollection* también implementa la interfaz *Storable* (ver sección 2.3.2).

Finalmente, el método *CiphertextCollection.shuffle_with_proof* realiza el proceso de mezcla verificable en la colección, devolviendo una tupla de la forma *(collection,proof)*, donde *collection* tiene tipo *CiphertextCollection* y corresponde a la mezcla de la colección original, mientras que *proof* es un objeto de clase *ShufflingProof*, codificando la prueba en conocimiento cero de que la colección original y *collection* son semánticamente equivalentes (ver sección 4.2.3, así como la discusión sobre *ShufflingProof* más adelante).

CiphertextReencryptionInfo

CiphertextReencryptionInfo representa la información necesaria para recifrar un texto cifrado en ElGamal, así como para verificar que cierto texto cifrado c' es un recifrado, generado usando tal información, del texto cifrado c. Cada instancia de *CiphertextReencryptionInfo* codifica una serie de recifrados $Renc_{r_x}(p, g, k_e, c_x)$ (ver 4.2.1) con parámetros aleatorios r_x específicos; uno por cada bloque c_x de un determinado texto cifrado, representado como una instancia de la clase *Ciphertext*. Actualmente, *CiphertextReencryptionInfo* codifica el recifrado para cada bloque como una pareja de valores $v_x = (g^{r_x} \ mod\ p, (k_e)^{r_x} \ mod\ p)$, en lugar de guardar r_x directamente. Cada bloque del texto cifrado $c_x = (\gamma, \delta)$ puede ser recifrado multiplicándolo módulo p por v_x. Esta codificación presenta algunas ventajas, como disminuir el tiempo que toma aplicar un recifrado a un cierto texto cifrado. Mas presenta también desventajas, como el uso del doble de espacio para almacenar la información del recifrado. Es posible que esta codificación cambie en versiones futuras de PloneVoteCryptoLib.

El constructor de *CiphertextReencryptionInfo* toma tan solo la llave pública *public_key*, con la cual debe estar cifrado el texto cifrado a recifrar. Este argumento debe ser un objeto de tipo *PublicKey* (ver sección 2.3.2). Dicha llave pública está disponible en adelante mediante el atributo *public_key* del objeto de recifrado.

Un objeto *CiphertextReencryptionInfo* así construido contiene cero bloques de información de recifrado. Es posible añadir este tipo de bloques manualmente como tuplas de la forma de v_x, mediante el método *add_block*. Sin embargo, existe una mejor forma de construir un objeto *CiphertextReencryptionInfo*: el método de clase *CiphertextReencryptionInfo.new*.

CiphertextReencryptionInfo.new es un método fábrica que toma la llave pública y longitud en bloques de un texto cifrado a recifrar y construye un objeto de recifrado *CiphertextReencryptionInfo*, listo para ser aplicado al mismo texto cifrado.

El método *apply* de *CiphertextReencryptionInfo* permite aplicar un recifrado a un texto cifrado compatible (con la llave pública y tamaño esperados), para generar otro objeto *Ciphertext* que es un recifrado del texto cifrado dado. Generalmente, el proceso de construir un recifrado de un cierto texto cifrado usando esta clase consta de dos pasos: 1) generar un recifrado con los parámetros apropiados mediante *new*; 2) aplicar tal recifrado al texto cifrado mediante *apply*.

Como discutimos en la sección 4.2.1, dados dos textos cifrados c y c' en ElGamal, donde c' es un recifrado de c, es imposible verificar de forma eficiente que realmente codifican el mismo texto claro, sin contar con información adicional. Esto es cierto también para dos objetos *Ciphertext*, creado uno mediante la aplicación de un objeto *CiphertextReencryptionInfo* al otro. Sin embargo, dado el objeto *CiphertextReencryptionInfo* codificando tal recifrado, el método *verify* de este objeto permite verificar que el texto c' es un

recifrado de c. Tal método devolverá verdadero si su segundo argumento es un recifrado de su primer argumento utilizando la información almacenada en el objeto de recifrado, y devolverá falso en otro caso.

CiphertextReencryptionInfo es indizable e iterable como una colección de valores v_x de recifrado, implementando los métodos especiales de python $\_\_iter\_\_$ y $\_\_getitem\_\_$.

CiphertextCollectionMapping

CiphertextCollectionMapping representa la transformación dada por el proceso de mezcla por recifrado entre dos colecciones de textos cifrados, incluyendo la información de los recifrados para cada elemento de la colección original en un elemento de la colección resultante (como objetos *CiphertextReencryptionInfo*), así como la permutación entre los elementos de ambas colecciones.

La forma más sencilla de construir un nuevo objeto de clase *CiphertextCollectionMapping*, y la única soportada por la interfaz pública de la clase, es mediante el método fábrica *new*. Tal método toma un objeto de tipo *CiphertextCollection*, correspondiente a la colección que deseamos mezclar. *new* no genera la colección mezclada, pero sí genera una transformación *CiphertextCollectionMapping* apropiada para tal colección, tomando en cuenta el número y longitud de sus elementos y la llave pública de la colección.

La colección mezclada puede entonces ser generada, aplicando la transformación a la misma colección para la cual fue creada, mediante el método *apply* de *CiphertextCollectionMapping*. Este método toma un objeto de tipo *CiphertextCollection* y devuelve otro que representa el resultado de aplicar la transformación a dicha colección.

El método *verify* de *CiphertextCollectionMapping* toma dos colecciones de textos cifrados, y verifica que la segunda sea obtenida de la primera mediante la transformación codificada en la instancia actual de *CiphertextCollectionMapping*. Claramente, esta verificación no es en conocimiento cero, puesto que *CiphertextCollectionMapping* contiene toda la información necesaria para relacionar entre sí los elementos de las dos colecciones.

El método *CiphertextCollectionMapping.rebase* proporciona la operación de cambio de origen entre transformaciones de colecciones de textos cifrados. Esta operación es utilizada en la construcción de la prueba en conocimiento cero representada por *ShufflingProof*. Dadas tres colecciones A, B y C, junto con las transformaciones $A \to B$ y $A \to C$, codificadas cada una como un objeto *CiphertextCollectionMapping*, *rebase* permite obtener la instancia de *CiphertextCollectionMapping* que codifica la transformación necesaria para obtener B a partir de C. Es decir, $(A \to B).rebase(A \to C)$ regresa una instancia de *CiphertextCollectionMapping* codificando la transformación $C \to B$.

ShufflingProof

ShufflingProof es, como ya mencionamos, la clase que representa la prueba de equivalencia semántica en conocimiento cero entre dos colecciones de textos cifrados.

El método fábrica *ShufflingProof.new* toma dos objetos *CiphertextCollection*, junto con el objeto *CiphertextCollectionMapping* que da la transformación entre ambos (con toda la información de la relación entre sus elementos), y devuelve una instancia de *ShufflingProof*. Dicha instancia contiene múltiples objetos *CiphertextCollection*, cada una representando una nueva mezcla de la colección original, al modo de las colecciones $\overline{C_{j1}}, ..., \overline{C_{j\alpha}}$ de la sección 4.2.3. Dado el reto φ generado mediante la aplicación de la función de hash SHA256 al conjunto formado por la colección original *original_collection*, la colección mezclada *shuffled_collection* y las colecciones $\overline{C_{j1}}, ..., \overline{C_{j\alpha}}$, la instancia de *ShufflingProof* contiene también:

1. Para cada β tal que $\varphi[\beta] = 0$, la transformación (como un objeto *CiphertextCollectionMapping*) entre *original_collection* y $\overline{C_{j\beta}}$.

2. Para cada β tal que $\varphi[\beta] = 1$, la transformación (como un objeto *CiphertextCollectionMapping*) entre $\overline{C_{j\beta}}$ y *shuffled_collection*.

Esto es, *ShufflingProof* codifica internamente los valores de la prueba $\wp$ de equivalencia semántica de colecciones de textos cifrados en conocimiento cero, dada en la sección 4.2.3, junto con las colecciones $\overline{C_{j1}}, ..., \overline{C_{j\alpha}}$ necesarias para verificar tal prueba.

Esos valores no son accesibles mediante la interfaz pública de *ShufflingProof*. En cambio, esta clase provee el método *verify*, que toma de nuevo la colección original y la colección mezclada, y verifica la prueba codificada por la instancia actual de *ShufflingProof*, siguiendo el algoritmo 4.3, con *original_collection* como C_j y *shuffled_collection* como C_{j+1}. Si este método regresa verdadero, esto quiere decir que ambas colecciones son semánticamente equivalentes[12].

La instancia de *ShufflingProof* devuelta por *new* no guarda información adicional que permita relacionar los elementos de *original_collection* con *shuffled_collection*. Una vez descartada la transformación directa usada para obtener una de la otra, tenemos solamente una prueba en conocimiento cero de su equivalencia, dada por *ShufflingProof*.

ShufflingProof implementa la interfaz *Storable*.

El valor α está dado como uno de los parámetros del módulo params.py (ver sección 2.3.2.3). En particular, *params.MIX_PROOF_SECURITY_PARAM* da la α usada para generar nuevas pruebas mediante *ShufflingProof.new*, mientras que *params.MIX_PROOF_SECURITY_PARAM_MIN* da la mínima α tal que un objeto *ShufflingProof* con esta α como parámetro de segu-

[12]Salvo con una probabilidad muy baja ($k_1/2^\alpha$, con k_1 constante) de que la prueba haya podido ser falsificada.

ridad será aceptado como válido (por ejemplo, al ser cargado desde un archivo). Ambos parámetros dependen a su vez de *params.SECURITY_LEVEL* por omisión, aunque pueden ser variados independientemente también, mediante *CUSTOM_MIX_PROOF_SECURITY_PARAM* y *CUSTOM_MIX_PROOF_SECURITY_PARAM_MIN*, respectivamente.

4.3.2.1. Soporte para TaskMonitor en plonevotecryptolib.Mixnet

Varios métodos de las clases dentro de **plonevotecryptolib.Mixnet** pueden tomar un tiempo comparativamente largo en ejecutarse. Consecuentemente, estos métodos permiten observar su progreso mediante la clase *TaskMonitor* (ver sección 2.3.2.1).

Estos métodos toman un parámetro opcional con nombre[13] *task_monitor*, de tipo *TaskMonitor*. Por ejemplo, *shuffle_with_proof* puede ser invocado en cualquiera de las siguientes dos formas:

```
CiphertextCollection.shuffle_with_proof()
CiphertextCollection.shuffle_with_proof(task_monitor=my_tm)
```

En el segundo caso, reportará su progreso de forma asíncrona a través del *TaskMonitor* dado como argumento.

Además de *shuffle_with_proof*, los métodos *new*, *apply* y *verify* de las clases *CiphertextReencryptionInfo*, *CiphertextCollectionMapping* y *ShufflingProof*, así como *CiphertextReencryptionInfo.subtract* y *CiphertextCollectionMapping.rebase*, aceptan un objeto *TaskMonitor* como argumento opcional.

[13] *keyword parameter*, en python.

Capítulo 5

SVE, PloneVote y PloneVoteCryptoLib

5.1. Elecciones Simples Verificables

El sistema PloneVote, descrito brevemente en la introducción y a más detalle en la sección siguiente, basa su protocolo de votación en el esquema de Elecciones Simples Verificables (SVE, por sus siglas en inglés), publicado por Benaloh en [Benaloh2006].

SVE separa el proceso de votación en dos partes: el proceso de captura de voto, en que un usuario construye, codifica, cifra y envía su voto al sistema; y el proceso de conteo de votos, en el cual los votos son procesados y contados de forma verificable.

El proceso de captura de voto propuesto por Benaloh depende de equipo especializado, capaz de imprimir recibos seguros tras un cristal protector, y requiere que cada elector capture su voto en tal equipo. Aunque tal proceso de captura posee propiedades interesantes de seguridad, incluido un cierto nivel de protección ante coerción, no será utilizado por PloneVote. Como describiremos más adelante (secciones 5.2.2 y 5.2.3), PloneVote permite, en cambio, que cada usuario vote desde cualquier equipo de cómputo personal conectado a internet en que confíe, sacrificando dicha protección ante coerción por conveniencia para el elector y para la administración de la elección. En el escenario particular de elecciones intra-institucionales, consideramos éste un compromiso adecuado (ver sección 5.2.4.7).

De momento, supongamos tan solo que cada votante puede, de alguna forma, capturar sus preferencias como un voto cifrado con alguna llave pública en una instancia de ElGamal[1], y recibir un recibo correcto de tal

[1] En [Benaloh2006], se menciona que el esquema general de SVE puede ser construido sobre cualquier esquema criptográfico que soporte cifrado de umbral y recifrado verificable. Sin embargo, el único tal esquema que hemos revisado en esta tesis es ElGamal, el cual

voto. Este recibo es una huella digital del voto cifrado, obtenido mediante la aplicación de una función hash segura a dicho texto cifrado. Los votos son guardados, junto con sus correspondientes recibos, en algún servidor no necesariamente confiable.

Dado esto, describiremos el proceso de conteo verificable para estos votos presentado en [Benaloh2006]. Esta parte del esquema SVE sí es utilizada en PloneVote, como base de nuestro protocolo de la elección.

Antes que comience una elección en SVE, se debe elegir una comisión de la elección, formada por n miembros que se considera improbable estén coludidos entre sí. Se elige también un cierto umbral $t \leq n$. Dicho umbral debe representar una mayoría de los miembros en la comisión, por ejemplo 3 de 5 u 8 de 10. La comisión debe generar un esquema de cifrado de umbral t de n (Capítulo 3), en que cada miembro de la comisión posee una llave privada del esquema y la llave pública es puesta a disposición de todos los participantes en la elección. Hemos visto ya que es posible generar tales llaves de forma distribuida (sección 3.2.2.2), utilizando un servidor central no-confiable como medio de intercambio de mensajes (sección 3.3.1), y de forma que los miembros de la comisión puedan cada uno verificar que la llave pública generada realmente es una llave de umbral t de n que incluye su llave privada.

La llave pública generada por el proceso anterior será usada para cifrar los votos en el proceso de captura. Una vez capturados todos los votos, tenemos una colección de votos cifrados con la llave pública de umbral de la comisión, donde cada voto se encuentra asociado con su recibo y, posiblemente, con la identidad del elector que lo emitió. Deseamos remover la asociación entre voto y elector.

Anonimizamos los votos mediante una red de mezcla por recifrado públicamente verificable (Capítulo 4). Cada mezcla de la colección de votos es guardada, junto con su prueba de mezcla (sección 4.2.3), en el mismo servidor que la colección original. Los servidores de mezcla no están especificados por SVE, pero una posibilidad es pedir que al menos t miembros de la comisión de la elección mezclen los votos.

Una vez que suficientes mezclas han sido realizadas para confiar en que la privacidad de los electores será protegida, la comisión coopera para descifrar todos los votos capturados. El descifrado es realizado de forma distribuida (sección 3.2.2.1), con pruebas de descifrado parcial (sección 3.2.3). Tras ser descifrados, el texto claro de los votos es publicado, junto con los descifrados parciales y pruebas correspondientes.

Al finalizar la elección, cualquier usuario puede auditarla, verificando la correctitud del proceso de conteo de los votos. Supongamos de momento que el conjunto de votos capturados originalmente representa realmente los votos emitidos por los electores. Cualquier usuario puede entonces verificar una por una todas las pruebas de mezcla en la red de mezcla desde ese conjunto hasta

será usado por PloneVote.

el que fue finalmente descifrado. El usuario puede entonces comprobar que tal conjunto fue descifrado correctamente, verificando las pruebas adjuntas a cada descifrado parcial y combinando por él mismo dichos descifrados parciales. Finalmente, el usuario puede realizar el conteo de los votos en texto claro obtenidos mediante dicho descifrado verificable, y corroborar que el resultado de la elección corresponde al publicado oficialmente.

Así pues, cualquier usuario puede comprobar el conteo de votos desde la colección inicial de votos capturados hasta los resultados finales de la elección, con esencialmente total certeza[2] y sin necesidad de confiar en ninguna entidad distinta de sí mismo. Que los votos de la colección inicial realmente representan aquellos emitidos por los electores puede ser comprobado si cada elector revisa que su voto cifrado aparezca en tal colección. Para ello, el elector puede tomar la huella digital de cada voto en la colección y compararla con el recibo que generó durante el proceso de captura de voto.

La privacidad de la elección queda asegurada por la comisión de la elección. El esquema de umbral utilizado para cifrar cada voto fuerza a que al menos t miembros de la comisión deban cooperar para descifrar cualquier voto. Una comisión de la elección con no más de $t-1$ miembros deshonestos descifrará solo el último conjunto de votos, obtenido tras un número seguro de mezclas. Si tal número de mezclas debe incluir mezclas por t miembros de la comisión, entonces algún miembro honesto de la comisión debe haber realizado un proceso de mezcla sobre los votos antes de que éstos sean descifrados, lo que protege la privacidad del elector.

Es importante notar que ni las pruebas de mezcla ni las de descifrado parcial, permiten a ningún usuario descifrar una colección de votos que no fue descifrada por t miembros de la comisión, u obtener alguna asociación entre los votos de una colección previa a un proceso de mezcla con los de la colección resultante de dicho proceso. Es para garantizar lo anterior que es importante que dichas pruebas sean en conocimiento cero.

Finalmente, notemos que aunque SVE depende de la comisión de la elección para garantizar la privacidad del elector, ésta no juega ningún papel en garantizar la correctitud de la elección. Una elección en SVE puede ser demostrada correcta, si cada elector verifica la presencia de su voto emitido en la colección inicial de votos capturados y al menos un auditor honesto verifica el proceso de conteo desde dicha colección hasta los resultados finales; esto aun si todos los miembros de la comisión y el servidor que almacena la información de la elección, estuviesen coludidos para falsificar los resultados.

5.2. PloneVote

Como mencionamos en la introducción de este documento, el proyecto PloneVote es un trabajo en curso para construir un sistema de votaciones

[2]Salvo por la ínfima posibilidad de que alguna de las pruebas en conocimiento cero de mezcla o descifrado parcial hayan sido falsificadas exitosamente.

seguro y verificable por electores, destinado a su uso en elecciones en el Instituto de Matemáticas de la Universidad Nacional Autónoma de México, así como elecciones internas de otras instituciones. La biblioteca PloneVoteCryptoLib, presentada en este trabajo, constituye tan solo el primer paso para la implementación del sistema PloneVote.

El sistema constará de varios componentes principales, cada uno de ellos un programa de software completo, destinado a ejecutarse ya sea en un servidor central o en el equipo de cómputo personal confiable de alguno de los usuarios que participan en la elección. El componente de lado de servidor está pensado para ejecutarse dentro del entorno del manejador de contenidos Plone [Plone], integrándose de forma efectiva con la infraestructura actualmente usada por el Instituto de Matemáticas.

El proyecto PloneVote se encuentra aún en una etapa temprana de desarrollo, y su construcción queda fuera del alcance de este trabajo. Sin embargo, consideramos importante dedicar esta sección a presentar el sistema tal como está previsto actualmente, con el fin de mostrar el uso esperado de PloneVoteCryptoLib en la práctica.

Durante el proceso de definición de requerimientos para PloneVote se identificaron tres responsabilidades principales para el sistema:

- *Seguridad de la elección*: El sistema debe proveer los mecanismos necesarios y seguir los protocolos adecuados, de forma que las propiedades de seguridad de la elección, definidas en los requerimientos del sistema, se cumplan. Listamos las propiedades de seguridad previstas por el diseño de PloneVote en la sección 5.2.4, junto con los supuestos bajo los cuales cada propiedad debería ser garantizada por el sistema.

- *Esquema de votación*: El sistema debe dar soporte a los esquemas de representación y conteo de votos previstos para las elecciones internas del Instituto de Matemáticas (e.g. multivoto). Esto incluye la forma de representar el voto emitido por cada elector, la interfaz dada para capturar tal voto y el esquema de conteo final de dichos votos. PloneVote contempla un diseño extensible, en que el esquema de representación y conteo de votos puede ser reemplazado sin interferir con la seguridad de la elección o su control administrativo.

- *Control administrativo de la elección*: El sistema debe permitir, de forma flexible, configurar los parámetros administrativos de una elección, al momento de que ésta sea creada. Tales parámetros incluyen: lista de votantes autorizados, duración de las fase de votación, lista de candidatos u opciones de voto, etc[3].

[3] En el caso de PloneVote, los miembros de la comisión y su umbral, así como el esquema de representación y conteo de votos, también son parámetros configurables al inicio de la elección, pero interactúan más con las otras dos responsabilidades que con los requisitos administrativos.

En el diseño general de PloneVote, buscamos hacer estas responsabilidades, y los mecanismos usados para satisfacerlas, tan independientes entre sí como sea posible. PloneVoteCryptoLib tiene relación directa únicamente con la responsabilidad de seguridad de la elección.

En el resto de esta sección describiremos brevemente el sistema PloneVote completo. Sin embargo, en nuestra descripción haremos especial énfasis en la seguridad de la elección y en las propiedades del sistema que permiten cumplir con tal responsabilidad. Mencionaremos muy brevemente el trabajo contemplado para satisfacer las otras dos responsabilidades, y solo en la medida en que dicho trabajo se relaciona con la infraestructura de seguridad de PloneVote.

Una descripción más completa y balanceada de los detalles de PloneVote, tal como se contempla actualmente para su implementación, puede encontrarse en el documento de definición de requerimientos y el documento de arquitectura de dicho sistema, publicados junto con el código de PloneVote-CryptoLib.

5.2.1. Usuarios de PloneVote

PloneVote contempla cuatro roles distintos para los usuarios involucrados en una elección: administrador de la elección, miembro de la comisión de la elección, elector y auditor. Adicionalmente, nuestro modelo de seguridad contempla dos categorías de usuarios más que pueden tener algún efecto en la elección, aun cuando no son participantes legítimos en ésta: el administrador de la plataforma y los usuarios de acceso mínimo.

Un mismo usuario puede tener múltiples roles en una misma elección. Por ejemplo, los miembros de la comisión pueden ser electores también, o el administrador de la elección puede ser uno de los miembros de la comisión. PloneVote permite a un usuario cualquier combinación de roles. Sin embargo, se recomienda, para añadir transparencia y confianza a un proceso electoral, que los auditores sean externos a la administración de la elección, esto es, no sean ni el administrador de la elección ni miembros de la comisión.

Cada rol está definido con respecto a una elección particular. Un mismo usuario puede ser el administrador de la elección para una cierta elección A y ser solamente un elector más en una elección B distinta.

5.2.1.1. Administrador de la elección

El administrador de la elección es el usuario responsable de crear y configurar una nueva elección en el *Servidor de PloneVote* (sección 5.2.2.1).

El administrador de la elección es el encargado de seleccionar los parámetros administrativos de la elección, incluyendo: opciones de voto, lista de electores, duración de las fases de la elección, entre otros. El administrador de la elección debe también seleccionar a los miembros de la comisión y el número de éstos que deben cooperar para descifrar los votos al final de la

elección, así como el esquema de representación y conteo de votos usado para dicha elección.

En general, es tarea del administrador de la elección el especificar los valores de cualquier parámetro configurable que afecte una elección en PloneVote. Dichos parámetros son siempre hechos públicos a todos los usuarios del sistema, y sujetos a un proceso de revisión durante un determinado periodo antes del inicio de la elección. El administrador de la elección es también el encargado de resolver dudas y reclamaciones durante dicho proceso de revisión, posiblemente alterando la configuración de la elección cuando esto sea apropiado (e.g. un usuario cumpliendo las características pedidas por el reglamento oficial de la elección para votar no aparece en la lista de electores y debe ser añadido).

Una vez iniciada la fase de votación de la elección, el servidor de PloneVote ya no permite al administrador de la elección alterar la configuración de ésta. A partir de este punto, el administrador de la elección no obtiene privilegios especiales en PloneVote por tener este rol. La supervisión de la votación y el conteo de votos depende de la comisión de la elección, no del administrador.

5.2.1.2. Miembro de la comisión de la elección

Los miembros de la comisión de la elección actúan exactamente como los miembros de la comisión en SVE (ver sección 5.1). Son seleccionados al configurar la elección por el administrador de la elección, junto con un valor de umbral t.

Los miembros de la comisión cooperan para establecer un esquema de umbral t de n durante la fase de configuración de la elección. Durante la fase de conteo de la elección, estos miembros realizan mezclas verificables del conjunto de votos emitidos y, finalmente, cuando suficientes mezclas se han realizado, cooperan para descifrar los votos dentro del conjunto anonimizado por la red de mezcla.

Los miembros de la comisión utilizan el *Cliente de la Comisión de la Elección* (sección 5.2.2.2) para realizar sus acciones relacionadas con este rol.

5.2.1.3. Elector

Un elector es un usuario del sistema con derecho a votar en la elección actual.

El administrador de la elección define, durante la fase de configuración, la lista de electores para dicha elección. Esta lista es públicamente visible durante toda la elección y, antes de que inicie la fase de votación, es sujeta a revisión y reclamaciones por todos los usuarios del sistema. Al iniciar la fase de votación, aquellos usuarios que aparecen en la lista de electores tendrán el rol de elector en la elección.

En PloneVote, los electores podrán votar utilizando el *Cliente Web del Elector* (sección 5.2.2.3). Dicho cliente es una aplicación accesible desde la interfaz web del servidor de PloneVote, a través de un navegador de internet cualquiera compatible con estándares actuales de desarrollo web. Un elector no requiere estar familiarizado con PloneVote o ningún software adicional para poder votar en una elección, más allá de conocimientos básicos de cómo utilizar un equipo de cómputo personal y un navegador de internet.

5.2.1.4. Auditor

Un auditor es un usuario al cuál se le asigna la responsabilidad de verificar la integridad de la elección.

El hecho de que las elecciones realizadas sobre PloneVote sean públicamente verificables, permite a cualquier usuario tomar el rol de auditor, sin que éste le deba ser asignado por el administrador de la elección o ningún otro usuario. Sin embargo, es recomendable elegir de forma pública una comisión de auditores para cualquier elección cuyo resultado pueda tener un impacto significativo. Dichos auditores serán los encargados oficiales de comprobar la correctitud de la elección. Además de la comisión de auditores, cualquier usuario, entidad o grupo interesado, puede realizar su propia auditoría de forma independiente.

Es recomendable que los auditores designados oficialmente sean personas ajenas a la administración del proceso de la elección. Esto es, la comisión de auditores no debe incluir al administrador de la elección o a miembros de la comisión de la elección.

Tras la publicación de los resultados de una elección, los auditores (tanto oficialmente designados como voluntarios) utilizan el *Cliente del Auditor* (sección 5.2.2.4) para verificar la correctitud del proceso de conteo, incluyendo los procesos de mezcla, el descifrado de los votos y el conteo en sí de los votos descifrados.

5.2.1.5. Administrador de la plataforma

Un administrador de la plataforma es cualquier usuario con acceso privilegiado al servidor de PloneVote, donde dicho acceso le permite alterar el funcionamiento de los componentes del lado de servidor del sistema PloneVote, los archivos almacenados en el servidor o los valores de configuración de la elección usados por el servidor de PloneVote.

El administrador del servidor que ejecuta la instancia de Plone en la cual se ejecuta el servidor de PloneVote, así como cualquier usuario con acceso privilegiado a la base de datos de Zope o la propia instancia de Plone, puede ser considerado un administrador de la plataforma. Adicionalmente, cualquier atacante que gane acceso privilegiado a estos recursos de forma ilegítima (mediante un ataque informático) tendrá las mismas capacidades que los usuarios mencionados anteriormente y debe ser considerado también

como un administrador de la plataforma para propósitos de nuestro modelo de seguridad.

En el escenario esperado de una elección intra-institucional, es difícil garantizar la seguridad de la plataforma contra acceso ilegítimo con un nivel suficientemente alto de confianza. En la basta mayoría de los sistemas comerciales, simple acceso físico al sistema ejecutado el servidor de PloneVote es suficiente para alterar de forma arbitraria los datos contenidos en éste o el código ejecutado durante su operación. Más aún, el servidor ejecutando el servidor de PloneVote puede tener múltiples administradores a distintos niveles de su plataforma de software (un administrador del sistema operativo y otro de la instancia de Plone, por ejemplo), cualquiera de los cuales puede tener un interés en alterar los resultados de la elección.

Para la mayoría de los escenarios de uso de PloneVote, los procesos administrativos y directivas de seguridad necesarios para aumentar la fiabilidad de la plataforma a un nivel adecuado, que permita confiar en la correctitud de una elección, son prohibitivos en costo y esfuerzo. Además, estos procesos, por su propia naturaleza, nunca pueden garantizar con certeza absoluta que no existe alteración maliciosa del sistema a nivel de la plataforma, y son, por lo general, opacos a la mayoría de los usuarios, haciéndolos difíciles o imposibles de verificar.

PloneVote, siguiendo un protocolo de elecciones públicamente verificables, presenta una visión distinta del problema de seguridad de la plataforma. Nuestro sistema considera al administrador de la plataforma siempre como un posible atacante y, por tanto, no considera el servidor de PloneVote como un agente confiable[4]. El servidor de PloneVote puede presentar comportamiento arbitrario, incluyendo comportamiento malicioso, sin que la correctitud de la elección o la privacidad del elector se vea por ello comprometida, siempre y cuando otros usuarios del sistema actúen correctamente (en particular la comisión de la elección y al menos un auditor, ver la sección 5.2.4 para más detalles). Esto minimiza el impacto que un administrador malicioso de la plataforma puede tener sobre una elección en PloneVote.

Un administrador malicioso de la plataforma puede aún corromper la elección, pero dicha corrupción es fácilmente detectable por cualquier auditor. En la práctica, el mayor impacto que puede tener un administrador de la plataforma en una elección en PloneVote, consiste en impedir que los votos de dicha elección sean contados, anulando la elección (sección 5.2.4.5). Esto es imposible de prevenir, pues los votos son almacenados en el servidor de PloneVote, por lo que el administrador de la plataforma tiene, por definición, capacidad suficiente para alterar o eliminar dichos votos.

[4] Además de la discusión anterior, en la sección 3.1 ya argumentamos que construir un sistema de votación seguro suponiendo la existencia de un agente confiable es un problema trivial.

5.2.1.6. Usuario de acceso mínimo

Un usuario de acceso mínimo es cualquier usuario que puede interactuar con el servidor de PloneVote, sin poseer ningún rol adicional en la elección. Esto incluye a cualquier usuario con acceso de red al servidor que aloja la instancia de Plone en la cuál se ejecuta el servidor de PloneVote.

PloneVote permite a un usuario de acceso mínimo:

- Obtener cualquier información publicada por el servidor de PloneVote. Esta información incluye: la configuración general de la elección, la lista de electores, los votos cifrados, las pruebas de mezcla, las pruebas de descifrado parcial, los descifrados parciales y combinados de los votos y los resultados de la elección. En general, cuando decimos que alguna información es "publicada" en PloneVote, queremos decir que cualquiera de los usuarios del sistema, incluido el usuario de acceso mínimo, puede leer esta información.

- Adquirir el rol de auditor. Cualquier usuario de PloneVote, incluido el usuario de acceso mínimo, tiene la habilidad de realizar una auditoría de la elección. Esto incluye los privilegios suficientes para conectarse al servidor de PloneVote usando el cliente del auditor para auditar la elección de forma guiada.

PloneVote no permite al usuario de acceso mínimo añadir o modificar ninguna información relacionada con la elección. El único posible ataque contra una elección en PloneVote que puede ser realizado por un usuario de acceso mínimo, es uno de denegación de servicio por saturación contra el servidor de red que aloja la instancia del servidor de PloneVote. Técnicas comúnmente utilizadas de seguridad en sistemas de red pueden ser usadas para mitigar dicho ataque.

Pueden existir personas que no sean siquiera usuarios de acceso mínimo en PloneVote. Por ejemplo, el servidor puede ser alojado tras un cortafuegos que impide acceso a éste a aquellos usuarios que se encuentren fuera de la red interna de la institución en la cual se realiza la elección. En ese escenario, las personas fuera de la red interna no serán considerados como usuarios de PloneVote en absoluto y no pueden interactuar con la elección de ninguna forma (a no ser que exista un fallo en la seguridad del cortafuegos).

5.2.2. Programas constituyentes

El sistema PloneVote se compone de cuatro programas ejecutables distintos, cada uno ejecutándose en un contexto de seguridad diferente. Estos programas incluyen el componente de servidor de PloneVote, dos aplicaciones de escritorio y una aplicación web.

Esta división es por motivos de seguridad. Para obtener las propiedades descritas en la sección 5.2.4, requerimos ejecutar distintas operaciones en distintos dominios de seguridad, bajo el control de distintos usuarios.

Tal como argumentamos al evaluar trabajo previo realizado en elecciones en el Instituto de Matemáticas (ver sección 1.3.1), son pocas las garantías de seguridad que pueden ser obtenidas mediante un sistema que opera enteramente dentro de un mismo servidor. En particular, en tal escenario, el administrador de la plataforma (esto es, el usuario que controla dicho servidor) puede alterar arbitrariamente el protocolo de votación, incluyendo tanto captura como conteo de votos. Dado lo anterior, dicho usuario puede, por sí solo, alterar arbitrariamente los resultados de la elección, comprometer la privacidad de cualquier elector y falsificar cualquier tipo de verificación que se busque realizar sobre el sistema.

Para obtener elecciones seguras y verificables, es necesario que ciertas operaciones sean realizadas por ciertos usuarios utilizando agentes en los cuales confíen. PloneVote provee estos agentes como programas de cliente de código abierto, escritos en lenguajes interpretados cuyo código puede ser verificado inmediatamente antes de ejecutarse. Aun aquellos usuarios que no cuentan con el conocimiento técnico para verificar directamente estos programas, pueden ganar confianza en los mismos de varias formas distintas, incluyendo auditorías por grupos externos que verifiquen y firmen digitalmente dichos programas. En cualquier caso, código malicioso añadido a estos programas se encuentra a la vista, por lo que puede ser detectado mucho más fácilmente que una modificación a código ejecutándose en el servidor.

Es posible imaginar una modificación a PloneVote en que el protocolo de votación se realice nodo a nodo entre pares, sin intervención de un servidor central. Sin embargo, contar con un componente del sistema de lado del servidor presenta también varias ventajas. Primero, facilita el uso e implementación del sistema, proveyendo un repositorio central donde almacenar la información 'oficial' de la elección, de tal modo que pueda ser rápidamente accedida y auditada. Segundo, da mayor flexibilidad en el tiempo para realizar las acciones del protocolo, ya que no es necesario esperar a ningún par para sincronizar en acciones simples, requiriendo, en cambio, tan sólo comunicarse con el servidor, que se encuentra siempre en línea. Finalmente, mejora la robustez de nuestro sistema, ya que un servidor funcionando correctamente puede comprobar, durante la elección, la correctitud de varias de las acciones de los usuarios y detectar corrupción, intencional o accidental, de la información de la elección[5].

Tenemos, entonces, dos dominios de seguridad en los cuales se ejecutan distintos componentes de PloneVote:

- Servidor: El dominio de seguridad de servidor corresponde a todo aquel código que es ejecutado dentro del servidor que aloja la instancia de

[5]Cualquier tipo de corrupción a los resultados de la elección será detectada en la fase de auditoría (sección 5.2.3.4), independientemente de si el servidor opera correctamente o no. Sin embargo, una detección tan tardía podría invalidar la elección, mientras que las verificaciones en tiempo real de un servidor de PloneVote correcto permiten corregir el problema en cuanto éste ocurra y continuar con la elección sin invalidar sus resultados.

Plone sobre la cuál se ejecuta el servidor de PloneVote. Consideramos que este código actúa en nombre de un agente, potencialmente no-confiable, conocido como "el servidor de PloneVote". Dicho agente ejecuta las acciones designadas por los programadores del sistema PloneVote[6], cuando todos los administradores de la plataforma son no-maliciosos. Sin embargo, puede ejecutar, en cambio, las acciones designadas por cualquier administrador malicioso de la plataforma. Por lo tanto, el código de servidor puede ser considerado sólo tan confiable como el conjunto de los usuarios con acceso privilegiado, legítimo o no, a dicho servidor.

- Cliente: El dominio de seguridad de cliente corresponde al código que es ejecutado en el equipo de cómputo personal de algún usuario de PloneVote. Se considera que dicho código actúa en nombre del usuario que lo ejecuta. Esto presupone que cada usuario es dueño y tiene control absoluto sobre el equipo de cómputo que utiliza para ejecutar este código, y confía en el código de cliente utilizado (ya sea porque confía en la fuente de la cual lo obtuvo, en algún auditor independiente de dicho código o porque el propio usuario lo ha inspeccionado). Un usuario con cierto nivel de conocimientos técnicos puede alterar el funcionamiento de su propio código de cliente, por lo que otros participantes en el protocolo de votación (incluyendo el servidor), deberán confiar en dicho código sólo en la medida en que confíen en el usuario que lo ejecuta.

A continuación, describiremos brevemente cada uno de los programas constituyentes de PloneVote.

5.2.2.1. Servidor de PloneVote

El servidor de PloneVote está conformado por una serie de productos de Plone (extensiones al sistema base), los cuales proveen una interfaz web para la configuración, creación y supervisión de elecciones en PloneVote, así como una serie de servicios web que permiten al resto de los programas de PloneVote comunicarse con este servidor para realizar sus respectivas acciones conforme al protocolo de la elección.

El código del servidor de PloneVote se ejecuta en el dominio de seguridad de servidor.

El administrador de la elección usa directamente la interfaz web del servidor de PloneVote para crear y configurar una nueva elección, así como para realizar los cambios necesarios en respuesta a reclamaciones por parte de otros usuarios durante la fase de configuración. Opcionalmente, el servidor puede proveer también de un mecanismo para procesar dichas reclamaciones de forma estructurada.

[6] y verificadas independientemente por cualquier persona interesada, al ser PloneVote de código abierto.

La interfaz web del servidor debe mostrar la configuración pública de cada elección realizada sobre éste (incluyendo la lista de electores). Tras la votación, mostrará también las huellas digitales de los votos originalmente capturados, permitiendo a cada elector verificar que su voto aparece en la colección de votos previa a la red de mezcla, y a los auditores comprobar que dicha lista corresponde a los votos contados. Asimismo, al terminar la fase de conteo, el servidor debe publicar los resultados de la elección.

La información necesaria para verificar el conteo de votos, incluyendo las mezclas y descifrados parciales verificables, no es mostrada en la interfaz web, mas debe ser ofrecida por el servidor mediante la interfaz de servicio web esperada por el cliente del auditor.

El protocolo detallado en la sección 5.2.3 muestra las interacciones de los usuarios de PloneVote con el servidor, tanto mediante su interfaz web como mediante la comunicación de los distintos programas cliente con los servicios ofrecidos por éste.

5.2.2.2. Cliente de la Comisión de la Elección

Este es el cliente utilizado por los miembros de la comisión de la elección para generar las llaves del esquema de umbral en el que participan, mezclar una colección de votos y producir descifrados parciales de los votos dentro de la última colección mezclada.

El cliente de la comisión de la elección es un programa de escritorio multiplataforma[7], escrito en python. Dicho programa se ejecuta en el dominio de seguridad de cliente, para el miembro de la comisión que lo utiliza, y se comunica con el servidor de PloneVote siguiendo el protocolo descrito en la sección 5.2.3.

Este cliente almacena la llave privada de umbral[8] de cada miembro de la comisión en el equipo de cómputo personal de dicho usuario. Esto requiere que los miembros de la comisión utilicen siempre el mismo equipo de cómputo para interactuar con una elección particular. No existen planes, para la versión inicial de PloneVote, de permitir migrar tales llaves de forma sencilla entre equipos de cómputo.

5.2.2.3. Cliente Web del Elector

El cliente web del elector es utilizado por éste para emitir su voto. El proceso de captura y cifrado de voto, incluyendo la generación de un recibo de voto verificable, es realizado enteramente dentro del cliente web del elector, antes de que dicho voto sea transmitido al servidor de PloneVote.

El cliente web del elector es accesible como una aplicación web desde el servidor de PloneVote. Para el elector, el cliente web se comporta como

[7] Soporte para Windows XP en adelante, Mac OS 10.2 en adelante y distribuciones actuales de Linux, está contemplado en los requerimientos del sistema.

[8] Así como la llave privada 1-a-1 utilizada para establecer canales privados virtuales para la generación del esquema de umbral (ver secciones 3.3.1 y 5.2.3.1).

si fuera parte del sitio web que aloja la elección en la cuál desea votar. Sin embargo, por razones de seguridad y transparencia, el cliente web del elector se ejecuta enteramente en el dominio de seguridad del cliente, mediante código en JavaScript/ECMAScript que es interpretado por el navegador del usuario y actúa enteramente dentro del entorno de dicho navegador.

El diseño anterior fue escogido buscando un balance entre la facilidad de uso del sistema para el elector y la posibilidad de auditar correctamente el proceso de captura de votos. El cliente web permite al elector participar en la elección sin necesidad de instalar software adicional en su equipo de cómputo personal o aprender a usar una herramienta distinta de su navegador web habitual. Este cliente debe ser cuidadosamente diseñado para presentar una interfaz intuitiva, que no requiera entrenamiento adicional para el elector[9]. Por otro lado, al ejecutarse en el dominio de seguridad del cliente, un auditor puede siempre revisar que el código que conforma dicho cliente codifique un comportamiento correcto. Un auditor puede comprobar la correctitud del código mediante inspección manual, o simplemente revisando que la huella digital de los archivos .js obtenidos desde el servidor corresponda con la publicada para una versión de PloneVote previamente verificada por alguna entidad en la que confíe.

Para facilitar tal auditoría, el cliente web del elector es descargado en su totalidad al equipo de cómputo personal del usuario, junto con los archivos usados para la elección particular en la que se desea votar, antes de que el servidor conozca la identidad del usuario que lo utiliza. Solo al recibir el voto cifrado, como el último paso del proceso de captura, debe el servidor autentificar al usuario. Esto impide que el servidor utilice información sobre la identidad del usuario para distinguir cuando éste es tan solo un elector normal o cuando se trata de un auditor, entregando un cliente web distinto en cada caso.

5.2.2.4. Cliente del Auditor

Este es el cliente utilizado por los auditores para verificar los resultados de la elección en la fase de auditoría (5.2.3.4).

Al igual que el cliente de la comisión de la elección, el cliente del auditor es un programa multiplataforma escrito en python, el cual se ejecuta en el dominio de seguridad del cliente correspondiente al auditor que lo utiliza. Para realizar sus funciones, el cliente del auditor se comunica con el servidor de PloneVote mediante los servicios web ofrecidos por éste, siguiendo el protocolo descrito en la sección 5.2.3.

El cliente del auditor es usado para comprobar el proceso completo de conteo de votos, incluyendo los procesos de mezcla de la colección de votos capturados, así como los descifrados parciales de cada voto, su correspondencia con los votos descifrados publicados y el conteo final de dichos votos.

[9] Suponiendo que dicho elector ya está familiarizado con otras aplicaciones web comunes y con el patrón de interfaz de usuario asistente ("wizard" en inglés) [Tidwell2006, 17].

Durante la fase de votación, un auditor podrá también utilizar este cliente para descargar y verificar, de forma automatizada, el cliente web del elector provisto por el servidor de PloneVote. Esto debe ser posible sin que el servidor pueda distinguir el acceso iniciado por el cliente del auditor de un acceso normal al cliente web realizado por un elector real.

5.2.3. Elecciones en PloneVote

En las secciones anteriores describimos los participantes de una elección en PloneVote (sección 5.2.1), así como los componentes ejecutables que constituyen nuestro sistema (sección 5.2.2). A continuación, daremos el protocolo seguido por tales entidades, tanto usuarios como programas, para llevar a cabo una elección en PloneVote.

Una elección en PloneVote se divide en cuatro fases: configuración, votación, conteo y auditoría. Separaremos nuestra descripción del protocolo por fases, describiendo brevemente cada fase antes de pasar a listar los pasos seguidos por los usuarios y componentes de PloneVote durante la misma.

5.2.3.1. Fase de configuración

Durante la fase de configuración, la información necesaria para llevar a cabo la votación es ingresada al servidor de PloneVote, publicada por éste, y sujeta a un proceso de revisión por todos los usuarios del sistema. La información es corregida en respuesta a reclamaciones legítimas del electorado, si las hubiera. Finalmente, al pasar de esta fase a la fase de votación, los parámetros de configuración de la elección se consideran, en adelante, inmutables.

Es también durante esta fase que la comisión de la elección debe establecer el esquema de umbral que será usado para descifrar los votos emitidos (ver Alg. 3.3).

A continuación listamos los pasos realizados durante esta fase de la elección:

1. El administrador de la elección crea un objeto *Elección* en el servidor de PloneVote, mediante la interfaz web de Plone. El administrador debe proporcionar la siguiente configuración inicial de la elección:

   a) Nombre.

   b) Descripción.

   c) Esquema de representación y conteo de votos a ser usado.

   d) Numero de miembros de la comisión de la elección, así como el número (umbral) de éstos requerido para descifrar los votos.

   e) Duración de la fase de votación.

2. El administrador de la elección configura la boleta de la elección, la cual depende del esquema de representación de votos utilizado. Esta configuración incluye la lista de candidatos u opciones posibles para la elección.

3. El administrador de la elección configura la lista de electores. Dichos usuarios son elegibles dentro de la lista de usuarios de Plone y pueden ser seleccionados en grupo, con base en la información almacenada en el manejador de contenidos.

4. El administrador de la elección configura la lista de miembros de la comisión. Dichos usuarios son elegibles dentro de la lista de usuarios de Plone.

5. El servidor de PloneVote publica la configuración de la elección y notifica, mediante correo electrónico, a todos los usuarios involucrados, con un mensaje distinto para electores, candidatos y miembros de la comisión.

6. Cada candidato tiene la opción de aceptar o rechazar su candidatura. Esta acción es realizada mediante la interfaz web del servidor de PloneVote. Un candidato no gana ningún privilegio adicional en el sistema por ser candidato.

7. Cada miembro de la comisión, que reciba notificación del servidor de PloneVote, debe usar el cliente de la comisión de la elección para generar un par de llaves uno a uno, el cual será usado para el establecimiento de un canal privado virtual de comunicación sobre el servidor de PloneVote. Para ello, el miembro de la comisión:

   a) Ejecuta el cliente de la comisión de la elección

   b) Ingresa en éste la dirección web del servidor de PloneVote, junto con su nombre de usuario y contraseña

   c) Elige la opción de "Establecer canal seguro"[10], en respuesta a la cual el cliente de la comisión genera un nuevo par de llaves de ElGamal estándar, almacenando la llave privada localmente y registrando la llave pública en el servidor de PloneVote.

   d) Observa y guarda de forma segura la huella digital del canal (la huella digital de su llave pública). El usuario debe transmitir dicha huella digital a los demás miembros de la comisión, preferentemente de forma personal o, si esto no es posible, por un canal externo al sistema como correo electrónico extra-institucional.

[10] Los nombres de todas las opciones presentadas a los distintos usuarios de PloneVote están sujetos a cambios con base en futuros estudios de usabilidad.

8. Una vez que todos los miembros de la comisión hayan realizado el paso anterior, cada miembro de la comisión debe conectarse de nuevo al servidor, mediante el cliente de la comisión, y realizar el primer paso del establecimiento del esquema de umbral para la elección. Para ello, el miembro de la comisión:

   a) Ejecuta el cliente de la comisión de la elección.

   b) Ingresa en éste la dirección web del servidor de PloneVote, junto con su nombre de usuario y contraseña.

   c) Selecciona la elección actual de una lista de elecciones mostradas por el cliente de la comisión. Dicha lista es obtenida del servidor de PloneVote e incluye todas las elecciones en fase de configuración en las cuales el usuario actual es miembro de la comisión de la elección.

   d) Elige la opción "Establecer seguridad de la elección - Fase 1".

   e) Comprueba que los nombres y huellas digitales de los "canales seguros" de los demás miembros de la comisión, presentados por el cliente de la comisión y obtenidos del servidor de PloneVote, sean correctos. En caso de aceptar estos valores, el cliente de la comisión generará el compromiso del usuario actual para el esquema de umbral y lo enviará al servidor[11].

9. Una vez que todos los miembros de la comisión hayan realizado el paso anterior, el servidor de PloneVote genera la llave pública general de la elección y la publica junto con su huella digital.

10. Los miembros de la comisión deben usar una vez más el cliente de la comisión para conectarse al servidor y generar sus respectivas llaves privadas de umbral. Para ello, cada miembro:

    a) Ejecuta el cliente de la comisión de la elección.

    b) Ingresa en éste la dirección web del servidor de PloneVote, junto con su nombre de usuario y contraseña.

    c) Selecciona la elección actual de una lista de elecciones mostradas por el cliente de la comisión.

    d) Elige la opción "Establecer seguridad de la elección - Fase 2". Esto causa que el cliente de la comisión descargue todos los compromisos de los demás miembros de la comisión, tomando el propio para el usuario actual desde almacenamiento local, y los combine para generar su llave privada de umbral y la llave pública general

[11] En caso de que los valores no coincidan con las huellas digitales comunicadas personalmente al usuario por los demás miembros de la comisión, éste deberá inmediatamente notificar a las autoridades correspondientes de la irregularidad, para que ésta sea investigada. La elección no puede proseguir en este caso.

de la elección. La llave privada es almacenada en el equipo de cómputo personal del miembro de la comisión.

e) Observa y guarda de forma segura la huella digital de la llave pública general de la elección generada por el cliente de la comisión. Debe verificar que ésta corresponda a la publicada por el servidor de PloneVote.

11. Se prevé un periodo de tiempo, después de realizado el paso 5, para la adjudicación de quejas y aclaraciones respecto a la elección de parte de cualquier usuario. El administrador de la elección podrá editar los parámetros de la elección durante este periodo. Notemos, sin embargo, que cualquier alteración a los miembros de la comisión de la elección, o al número necesario de éstos para descifrar los votos, requerirá que los pasos 7 a 10 sean realizados de nuevo.

12. Al término del periodo anterior, y si la comisión de la elección ha terminado con el paso 10, el administrador de la elección puede iniciar la fase de votación. Esta acción es realizada mediante la interfaz web del servidor de PloneVote y causa que dicho servidor envíe una notificación, mediante correo electrónico, a todos los participantes de la elección (electores, candidatos y miembros de la comisión). Dicha notificación detalla la configuración de la elección al momento de iniciar y la votación (la cual no puede ya ser alterada legítimamente), incluyendo la huella digital de la llave pública general de la elección.

13. Los auditores de la elección deben corroborar que la configuración publicada por el servidor se ajuste a los reglamentos aplicables a la elección en curso.

14. Los miembros de la comisión deben verificar, una vez más, que la huella de la llave pública general de la elección, publicada por el servidor, concuerde con la que éstos obtuvieron al final del paso 10.

Es importante notar que los pasos 7 a 10, realizados por la comisión de la elección, corresponden a los pasos 2 y 3 del "subprotocolo de cifrado de umbral" descrito en la sección 3.3.1. El paso 7, en particular, solo debe ser realizado una única vez por un mismo usuario. Si, en un futuro, este usuario forma parte de otras comisiones de la elección, en elecciones distintas puede usar el mismo canal seguro (mismo par de llaves) en dichas elecciones también, omitiendo el paso 7.

El cliente de la comisión de la elección debe proveer algún mecanismo para crear un nuevo canal seguro en caso de que el usuario pierda acceso a su llave privada (por ejemplo al cambiar de equipo de cómputo). Sin embargo, para aquellas elecciones en que dicho usuario sea miembro de la comisión y que se encuentren en la fase de votación o posterior, será ya imposible cambiar las llaves criptográficas usadas por la elección y este usuario no podrá participar en el proceso de descifrado.

5.2.3.2. Fase de votación

Esta es la fase en que cada elector emite su voto y el servidor captura la colección original de votos cifrados.

Como parte de la configuración de la elección, el administrador de la elección debió elegir un tiempo de duración para la fase de votación, usualmente de un par de días. El servidor terminará la fase de votación, dejando de aceptar nuevos votos, cuando dicho periodo haya transcurrido. No existe forma (suponiendo un servidor de PloneVote funcionando correctamente) de finalizar la fase de votación antes o después del tiempo indicado en la configuración de la elección. El servidor tampoco permite modificar los parámetros de configuración de la elección durante esta fase o las siguientes.

Solo dos acciones válidas son posibles en la fase de votación: la emisión de un voto por parte de un elector y la verificación del cliente web del elector por parte de un auditor.

Para emitir su voto, un elector debe seguir los siguientes pasos:

1. El elector entra al sitio web del servidor de PloneVote, sin autentificarse aún como usuario, y elige la elección en la que desea votar.

2. El servidor de PloneVote proporciona al elector el cliente web del elector, como una aplicación web a ser ejecutada dentro del navegador de este usuario. El servidor proporciona también los archivos requeridos por dicho cliente para la elección particular, incluyendo la boleta de la elección y los detalles de configuración del esquema de representación de votos.

3. El cliente web del elector presenta a éste una interfaz gráfica que le permite capturar su voto. Esta captura es realizada únicamente dentro del navegador del elector. El voto en texto claro jamás es enviado al servidor.

4. El elector captura su voto y confirma su elección dentro del cliente web.

5. El elector elige entonces la opción "Cifrar voto" del cliente web. Esto causa que dicho cliente cifre el voto (dentro del navegador del elector) con la llave pública general de la elección.

6. El cliente web presenta al elector un recibo de su voto, el cual consiste de una huella digital del voto cifrado. El cliente da al elector la opción de imprimir o guardar este recibo.

7. El elector guarda su recibo, si así lo desea, y elige la opción "autentificar como elector" del cliente web. El cliente web transfiere entonces el voto cifrado al servidor de PloneVote.

8. El servidor de PloneVote pide al usuario autentificarse mediante su nombre de usuario y contraseña.

9. El elector se autentifica con el servidor.
10. El servidor verifica que el usuario como el cual se autentificó el elector esté autorizado para votar en la elección correspondiente (y no haya votado ya). En caso contrario, el servidor muestra un mensaje de error. En caso afirmativo, muestra una última pantalla de confirmación, preguntando al usuario si está seguro que desea emitir su voto y mostrando la huella digital del mismo.
11. El elector confirma que desea emitir su voto y el voto queda capturado.

Por su parte, cada auditor de la elección realiza los pasos siguientes:

1. Ejecuta el cliente del auditor.
2. Proporciona la dirección del servidor de PloneVote que aloja la elección.
3. Selecciona la elección actual de una lista de elecciones presentadas y elige la opción "Auditar captura".
4. El cliente del auditor simula entonces las acciones de un elector anónimo hasta que el servidor de PloneVote le proporciona el código y elementos de configuración de la elección.
5. El cliente del auditor presenta entonces la siguiente información al auditor:
   a) La configuración usada por el cliente web del elector, permitiendo contrastarla con la publicada por el servidor al final de la fase de configuración. Esta información incluye la huella digital de la llave pública general de la elección, el esquema de representación de voto usado y la boleta presentada a los usuarios.
   b) Una huella digital del código del cliente web del elector, que permite verificarlo contra las huellas digitales de versiones de este cliente auditadas por terceros.
   c) Un breve mensaje, indicando si el cliente del auditor tiene en su base de datos interna la huella del cliente web del elector, marcada como correspondiente a un cliente web correcto (en general, si ambos clientes provienen de la misma versión de producción del sistema PloneVote, el cliente del auditor reconocerá al del elector).
6. Si la información anterior resulta incongruente con la publicada al final de la fase de configuración de la elección, o bien la huella digital del cliente web del elector no corresponde a un cliente web considerado confiable, el auditor debe advertir inmediatamente a las autoridades de la elección (y posiblemente al electorado). En este caso, la elección debe ser cancelada y el servidor que aloja la elección analizado, para diagnosticar el problema y buscar evidencia de posibles ataques de seguridad.

5.2.3.3. Fase de conteo

En esta fase se realiza la mezcla, descifrado y conteo de los votos capturados en la fase anterior.

Las operaciones de esta fase son principalmente realizadas por la comisión de la elección, como sigue:

1. Cada miembro de la comisión se conecta en un momento distinto al servidor de PloneVote, utilizando el cliente de la comisión, dando la dirección del servidor, su nombre de usuario y contraseña, y eligiendo la elección apropiada.

2. El miembro de la comisión elige la opción "Realizar mezcla de votos". Dicha opción solo está disponible si no existe ningún otro usuario realizando una mezcla de votos al mismo tiempo para esa elección.

3. El cliente de la comisión descarga la última colección de votos cifrados y realiza una mezcla verificable de éstos, enviando la colección resultante, junto con la prueba de mezcla, al servidor de PloneVote.

4. El servidor de PloneVote verifica que la prueba de mezcla sea correcta y acepta la nueva colección como la última colección en su lista para la elección actual. El servidor guarda una copia de todas las colecciones mezcladas y sus pruebas de mezcla correspondientes.

5. El cliente de la comisión muestra a su usuario una huella digital de la mezcla realizada, la cual actúa como un recibo de mezcla.

6. El cliente de la comisión no presenta la opción de descifrar los votos hasta que hayan sido realizadas al menos tantas mezclas de los votos como miembros de la comisión son necesarios para el proceso de descifrado[12].

7. Una vez que suficientes mezclas verificables hayan sido realizadas por miembros de la comisión, cada uno de éstos deberá conectarse de nuevo mediante el cliente de la comisión y elegir la opción de "Descifrar votos".

8. El cliente de la comisión presenta a cada miembro una lista de huellas digitales de las mezclas realizadas. Si el miembro de la comisión realizó una mezcla de los votos, deberá revisar que la huella digital correspondiente a tal mezcla aparece en la lista[13].

9. El cliente de la comisión genera entonces un descifrado parcial de la última colección disponible de votos y lo envía al servidor, junto con la prueba de descifrado parcial para cada voto.

[12] Esto es solo indicativo y no representa una medida de seguridad fuerte, ya que suficientes miembros de la comisión podrían descargar incluso la primera colección de votos capturados del servidor y usar un cliente modificado para descifrarlos en conjunto.

[13] Esta comprobación puede automatizarse si el cliente de la comisión guarda el recibo para cada mezcla realizada mediante dicho cliente.

10. El servidor verifica cada prueba de descifrado parcial y solo acepta dicho descifrado si la prueba es correcta.

11. Una vez que suficientes descifrados parciales correctos de la última colección de votos estén disponibles en el servidor, éste los combina para obtener los votos en texto claro. El servidor realiza un conteo de estos votos, publicando los resultados de la elección a través de su interfaz web.

12. Para dar por terminada esta fase, el servidor publica, mediante su interfaz web, las huellas digitales de todos los votos cifrados emitidos originalmente por los electores en la fase de votación, asociadas cada una al nombre del elector que emitió el voto. A partir de este punto, el servidor permite a los auditores descargar todas las colecciones mezcladas, descifrados parciales y pruebas asociadas con la elección.

Podríamos permitir también, en esta fase, que usuarios adicionales a los miembros de la comisión realicen mezclas de la colección de votos. No es claro, sin embargo, que permitir tal acción incremente la seguridad de la elección en ninguna forma contundente. Si tal opción es añadida, se vuelve necesario, en cambio, contar con algún mecanismo para evitar que la generación de mezclas espurias sea utilizada como un ataque de denegación de servicio contra el sistema, ya que solo una mezcla puede ser realizada a la vez. Una posibilidad es permitir que miembros de la comisión cancelen mezclas en progreso de usuarios que no forman parte de la comisión.

5.2.3.4. Fase de auditoría

En la fase de auditoría, cada usuario debe comprobar que su propio voto fue capturado correctamente durante la fase de votación. Asimismo, los auditores de la elección deben comprobar que el proceso de conteo fue realizado correctamente y los resultados publicados corresponden a los votos capturados inicialmente.

Cada elector puede verificar que su voto fue capturado correctamente simplemente corroborando que la huella digital de su voto, en la lista publicada junto con los resultados de la elección, corresponde a aquella que aparece en su recibo de voto. Idealmente, cada usuario realizará tal comprobación sin autentificarse con el servidor, haciendo difícil que dicho servidor genere listas distintas para usuarios distintos. Aquellos usuarios que no emitieron voto alguno en la elección, deberán también verificar que esto se refleje en la lista publicada con los resultados de la elección.

Por otra parte, cada auditor realiza los siguientes pasos para verificar el proceso de conteo:

1. Ejecuta el cliente del auditor.

2. Proporciona la dirección del servidor de PloneVote que aloja la elección.

3. Selecciona la elección actual de una lista de elecciones mostradas y elige la opción "Auditar conteo".

4. El cliente del auditor descarga entonces todas las colecciones de votos mezcladas, desde la original hasta la que fue finalmente descifrada, junto con las pruebas de mezcla. El cliente descarga también los descifrados parciales y sus respectivas pruebas de descifrado parcial, así como la configuración de la elección.

5. El cliente del auditor muestra la configuración de la elección a su usuario, pidiendo confirmación de que ésta corresponde a la publicada al final de la fase de configuración.

6. Si el auditor confirma la configuración, el cliente muestra entonces las huellas digitales de todos los votos capturados en la colección original. El auditor debe comprobar que esta lista corresponde a la publicada por el servidor.

7. Si el auditor confirma la lista de votos capturados, el cliente valida entonces cada mezcla realizada, verificando la correctitud de la prueba de mezcla. Si esta verificación falla para alguna mezcla, el cliente del auditor muestra una advertencia, indicando la mezcla incorrecta, y termina.

8. El cliente del auditor verifica entonces la prueba de cada descifrado parcial, comprobando que éste corresponde a un descifrado parcial correcto de la última colección de votos en la red de mezcla.

9. El cliente del auditor combina los descifrados parciales para obtener los votos en texto claro y realiza un conteo de éstos de acuerdo con el esquema de conteo declarado para la elección.

10. El auditor debe entonces revisar que los resultados mostrados por el cliente del auditor sean congruentes con los publicados por el servidor de la elección.

Si cualquier paso de la verificación da resultados inesperados, el auditor debe informar del problema a las autoridades de la elección y/o al electorado.

5.2.4. Seguridad de PloneVote

En la introducción de este documento vimos una serie de propiedades de seguridad que un sistema de votación ideal debería satisfacer (sección 1.2.1). En las secciones 5.2.4.1 a 5.2.4.7, notaremos cuáles de estas propiedades cumple el sistema PloneVote, tal como fue descrito en las secciones anteriores, y bajo qué condiciones o supuestos. Argumentaremos brevemente cada una de nuestras afirmaciones sobre la seguridad del sistema.

Antes de comenzar con tal discusión, debemos mencionar un supuesto requerido para casi todas nuestras afirmaciones de seguridad: el supuesto de seguridad del cliente. Suponemos que el software utilizado por cada usuario del sistema, en su equipo de cómputo personal, es confiable para dicho usuario y realiza las acciones pedidas por éste de forma correcta. Esto incluye los clientes del auditor y de la comisión de la elección, así como el cliente web del elector.

El supuesto de seguridad del cliente es equivalente a afirmar que todo programa ejecutándose en el dominio de seguridad del cliente (ver sección 5.2.2) efectivamente actúa como un agente del usuario que lo ejecuta. Notemos que nuestra definición de seguridad del cliente nos permite que un programa cliente actué incorrectamente de acuerdo con el protocolo esperado de la elección, siempre que dicho comportamiento sea conocido y aprobado por el usuario del programa. Informalmente: un cliente puede mentir al resto del sistema a nombre de su usuario, mas no mentir a su usuario.

En la sección 5.2.4.8 describiremos más a detalle las implicaciones de este supuesto y los casos en que puede o no cumplirse en nuestra implementación inicial planeada. Supondremos en la discusión de las secciones 5.2.4.1 a 5.2.4.7, que se cumple la seguridad del cliente.

5.2.4.1. Correctitud

Bajo el supuesto de seguridad del cliente, PloneVote ofrece fuertes garantías de correctitud, las cuales dependen únicamente de métodos estadísticos (las pruebas en conocimiento cero de mezcla y descifrado parcial) y del proceso de verificación de la elección (fase de auditoría), sin requerir siquiera que se cumplan los supuestos criptográficos del esquema ElGamal[14].

Provisto que cada elector emita su voto con un cliente correcto, ejecutándose en un equipo de cómputo personal confiable, éste obtendrá un recibo que le permite corroborar que su voto se encuentra en el conjunto original de votos capturados. Dicho recibo debe corresponder solamente al voto correcto emitido por el elector, ya que no existe forma práctica de generar un voto espurio que tenga el mismo recibo (generado mediante una función hash segura) que el voto original. Que la colección original de votos capturados representa correctamente las preferencias del electorado, queda entonces garantizado una vez que todos los electores comprueban su recibo contra las huellas digitales publicadas para los votos dentro de dicha colección.

Al verificar cada proceso de mezcla, el auditor puede asegurarse de que la colección final de votos que fueron descifrados es idéntica a la colección original de votos capturados. Verificando las pruebas de descifrado parcial, el mismo auditor puede corroborar que el descifrado de tal colección fue realizado correctamente, y que, por tanto, los votos en texto claro publicados por el servidor corresponden a los votos emitidos por los electores. Finalmente, el auditor realiza el conteo de estos votos, obteniendo los resultados

[14] e.g. la dificultad del problema de Diffie-Hellman.

correctos de la votación, los cuales puede comparar con los publicados por el servidor[15].

En consecuencia, si cada elector verifica su recibo contra la lista de recibos publicados por el servidor y al menos un auditor honesto verifica correctamente el proceso de conteo, la correctitud de la elección completa queda garantizada, salvo por la ínfima probabilidad de que algún miembro de la comisión haya podido falsificar correctamente una prueba de mezcla o descifrado parcial. Tal probabilidad es despreciable en la práctica si usamos un parámetro de seguridad adecuado para nuestras pruebas (e.g. $\alpha = 128$).

Incluso si toda la comisión de la elección es maliciosa, ésta no puede comprometer la correctitud del conteo de los votos si dicho proceso de conteo es verificado por al menos un auditor honesto.

5.2.4.2. Privacidad

En general, la privacidad del elector puede ser violada si y sólo si existe forma de asociar el voto cifrado emitido por éste con el texto claro de dicho voto.

Bajo el supuesto de seguridad del cliente, el cliente web del elector no revelará a ningún tercero el texto claro del voto del elector, tan solo el texto cifrado correspondiente a dicho voto. El cifrado de cada voto se realiza usando la llave pública general de la elección. Las acciones de la comisión de la elección, durante la fase de configuración, garantizan que se trata de una llave pública para un esquema de umbral, con los miembros de la comisión como participantes.

Suponemos la dificultad del problema de Diffie-Hellman y el uso de una instancia adecuada de ElGamal (e.g. con p suficientemente grande). Entonces, sólo un grupo de al menos t miembros de la comisión (donde t es el umbral elegido durante la configuración de la elección), actuando en conjunto, podrá descifrar un voto cifrado con la llave general de la elección. Una comisión de la elección con t miembros maliciosos, coludidos entre si, puede trivialmente violar la privacidad del elector: basta que dichos miembros cooperen para descifrar la colección original de votos previa a cualquier proceso de mezcla.

La privacidad del elector también se vería comprometida si algún atacante fuera capaz de rastrear su voto a través de las mezclas, desde la colección original hasta aquella que es descifrada para su conteo. Como t miembros de la comisión deben realizar mezclas de los votos antes de que éstos sean descifrados y una sola mezcla honesta basta para evitar la posibilidad de rastrear un mensaje a través de una red de mezcla, tal ataque requiere, de nuevo, de la colusión de t miembros de la comisión.

Así pues, a diferencia de la correctitud de la elección, la privacidad del elector sí depende de la seguridad del esquema criptográfico usado, así como

[15] El proceso que debe ser seguido por el auditor, aunque complejo, es automatizado en su mayor parte por el cliente del auditor.

de que al menos $n-t+1$ miembros de la comisión no se encuentren coludidos entre si para atacar dicha privacidad.

Es razonable dar por hecho el primer supuesto en la práctica, considerando el amplio análisis que ha recibido la seguridad de ElGamal, sin que ataques efectivos hayan sido descubiertos a la fecha. Por su parte, la no-colusión de la comisión puede ser obtenida, con certeza razonable para el elector, mediante una selección cuidadosa de sus miembros (representado cada uno intereses distintos) y el parámetro t del esquema de umbral.

Cumpliéndose lo anterior, un elector que vote sin ser observado, y utilizando un cliente correcto, puede tener confianza en que su voto nunca será revelado como tal a ningún otro usuario.

Como veremos en la sección 5.2.4.7, es posible para un elector revelar voluntariamente los contenidos de su voto a un tercero, sacrificando su privacidad.

5.2.4.3. Verificabilidad

Como vimos en la sección 5.2.4.1, cada usuario puede verificar que su propio voto fue capturado correctamente e incluido en la colección inicial de votos cifrados. Asimismo, cualquier auditor puede verificar el proceso de mezcla, conteo y descifrado, garantizando que los resultados de la elección corresponden a los votos en dicha colección inicial. Todo usuario del sistema puede opcionalmente tomar el rol de auditor, si así lo desea.

Decimos entonces que PloneVote cuenta con un proceso de captura de votos individualmente verificable: cada elector puede verificar que su voto fue capturado correctamente; y un proceso de conteo de votos universalmente verificable: cualquier participante en la elección puede verificar que todos los votos capturados fueron contados correctamente. Esto bajo las mismas condiciones necesarias para garantizar correctitud: seguridad del cliente e improbabilidad estadística de falsificar una prueba de mezcla o descifrado parcial con los parámetros de seguridad elegidos.

5.2.4.4. Justicia

Si bien la verificabilidad de PloneVote está relacionada con su propiedad de correctitud, su justicia está ligada a la propiedad de privacidad. Al ser imposible obtener el texto claro de los votos antes de que sea realizado el descifrado distribuido y conteo de la última colección mezclada, en la fase de conteo, será imposible obtener resultados parciales de la elección durante la fase de votación.

Suponiendo que todos o casi todos los electores mantienen su voto secreto, que ElGamal es un esquema criptográfico seguro, y que la comisión de la elección no incluye un subconjunto de t o más miembros coludidos para atacar la justicia de la elección, PloneVote garantiza esta propiedad.

5.2.4.5. Robustez

La robustez de PloneVote depende del servidor de PloneVote y de la comisión de la elección.

Un servidor malicioso o comprometido no es capaz de comprometer la correctitud o privacidad de una elección en PloneVote, pero sí puede comprometer su robustez, impidiendo el correcto seguimiento del protocolo de la elección y forzando la cancelación de dicha elección.

Cualquier ataque, incluido un ataque de denegación de servicio, que interrumpa el funcionamiento del servidor de PloneVote durante la fase de votación, puede detener la elección. Asimismo, ataques que puedan alterar los archivos guardados por el servidor, como las colecciones de votos cifrados, las pruebas de mezcla o los descifrados parciales verificables, pueden, en ciertos casos, ser suficientes para invalidar la elección. Aunque dichas alteraciones son fácilmente detectables mediante el proceso de verificación, la eliminación de votos puede hacer imposible obtener los resultados de la elección, y la eliminación de pruebas de mezcla y descifrado parcial puede evitar la validación de los resultados, aun cuando sea posible obtenerlos.

Técnicas comunes de seguridad en servicios de red deben ser utilizadas para proteger el servidor de PloneVote y el sistema en que éste se ejecuta contra ataques externos. Se debe considerar a cualquier administrador de la plataforma (ver sección 5.2.1.5), como suficientemente confiable para que sea improbable que dicho administrador busque bloquear la elección. En el escenario de elecciones institucionales, y dado que ataques contra la robustez del sistema son fácilmente detectables, consideramos éste un supuesto razonable.

Cualesquiera $n - t + 1$ miembros de la comisión de la elección pueden también frenar un proceso electoral, rehusando cooperar para descifrar el conjunto final de votos mezclados. El mismo efecto ocurre si tal número de miembros de la comisión, por cualquier razón, pierden acceso a sus correspondientes llaves privadas de umbral. Para minimizar el riesgo de esta eventualidad, se recomienda elegir t cuidadosamente, de tal modo que uno o dos miembros de la comisión puedan no participar en el descifrado final de los votos, sin evitar su realización.

5.2.4.6. Democracia

Claramente, cualquier ataque que afecte la robustez de PloneVote, afecta también la propiedad de democracia, impidiendo que uno o más usuarios voten en la elección afectada.

La lista de electores publicada durante la fase de configuración de la elección, junto con su proceso para resolución de reclamaciones, permite tener constancia de quién está o no autorizado a votar en una elección particular. Por su parte, la lista de recibos de votos publicada por el servidor registra qué electores emitieron su voto en la elección.

Se deben establecer procesos administrativos para atender reclamaciones en los siguientes casos:

- Algún usuario que debería aparecer en la lista de electores generada en la fase de configuración no aparece en ésta. Viceversa, algún usuario que no debería ser capaz de votar en la elección aparece en la lista de electores.
- El servidor de PloneVote impide capturar su voto a un usuario que aparece en la lista de electores.
- El voto de un elector que votó en la elección no aparece en la lista original de votos capturados.
- Aparece un voto en la lista original de votos capturados correspondiente a un usuario que no votó en la elección.

Dichos procesos quedan fuera del alcance de PloneVote como sistema de software.

Notemos que, salvo por el primer punto de nuestra lista, un servidor de PloneVote funcionado correctamente evita los problemas aquí descritos. Sin embargo, no contamos con una garantía más fuerte (por métodos estadísticos o criptográficos) de que la propiedad de democracia será cumplida por PloneVote.

Por otro lado, la verificabilidad de PloneVote sí garantiza que cualquier usuario, al que se le impida votar o cuyo voto no sea capturado correctamente, se dé cuenta de tal problema. Esto, aunado a un proceso administrativo correcto para atender reclamaciones, nos permite dar soporte confiable a la propiedad de democracia en elecciones en PloneVote.

5.2.4.7. Incoercibilidad

Por si mismo, usado como sistema para elecciones en línea, PloneVote no ofrece protección significativa contra coerción.

Cualquier persona que observe a un elector votar mediante el cliente web del elector y obtener su recibo de voto, puede usar posteriormente dicho recibo para verificar que el voto que observó aparece en la colección de votos capturados. Así pues, cualquier atacante que pueda forzar al elector a votar en su presencia puede conocer su voto.

Adicionalmente, un elector que desee vender su voto podría utilizar un cliente del elector modificado para generar una prueba de cifrado de su voto (e.g. el valor aleatorio r utilizado para cifrar el voto mediante el Alg. 2.4). Entregando tal prueba a un tercero, junto con el valor de su voto en texto claro, el elector puede demostrar que ése fue realmente el voto que emitió y que aparece en la colección original de votos capturados.

Notemos que, si el elector vota sin ser observado, el recibo que obtiene del cliente web del elector, sin modificar, no es suficiente para mostrar a un

tercero que votó de cierta forma. El recibo sólo contiene la huella digital del voto cifrado en la colección original de votos capturados en el servidor. Bajo los mismos supuestos necesarios para garantizar privacidad de la elección (seguridad de ElGamal y no-colusión de t miembros de la comisión), estos votos nunca serán descifrados directamente: su contenido será revelado para el conteo sólo después de múltiples procesos de mezcla que impiden asociar un voto especifico cifrado con su texto en claro.

En resumen, PloneVote permite coerción del elector al momento de emitir su voto, mas no posteriormente.

Podría pedirse que cada elector vote en una estación de captura físicamente protegida, usando un cliente web del elector correcto y vetado por varios auditores distintos. El proceso de auditoría de estas estaciones tendría que ser distinto al usado para el escenario de elecciones en línea que hemos considerado hasta el momento, ya que las estaciones no están bajo el control del elector y es necesario garantizar a éste que su voto es capturado correctamente. Esto daría al proceso de la elección una medida razonablemente fuerte de protección contra coerción, sin embargo, eliminaría la conveniencia obtenida por el elector al votar en línea desde cualquier equipo de cómputo personal. Además, asegurar confianza en las estaciones de captura complica el proceso de la elección, aumenta su costo e introduce una vía adicional de ataque contra su correctitud.

En el escenario de elecciones intra-institucionales, al cual está enfocado el sistema PloneVote, es razonable suponer que el riesgo de coerción del elector no es significativo. En tal escenario, las diferencias de poder entre el elector promedio y cualquier persona o grupo con un interés en manipular el resultado de la elección tienden a ser menores que, por ejemplo, en una elección nacional o estatal. Adicionalmente, procesos administrativos para reportar intentos de coerción pueden servir para disuadir de esta práctica, haciéndola excesivamente riesgosa en comparación con el beneficio que un atacante pueda ganar de ella. Siguiendo el razonamiento anterior, PloneVote será implementado para soportar emisión de votos en línea, sin considerar el uso de estaciones de captura físicamente seguras.

5.2.4.8. Supuesto de seguridad del cliente

Recordemos, de la discusión al principio de la sección 5.2.4, que nuestras propiedades de seguridad anteriores, incluyendo correctitud y verificabilidad, dependen del supuesto de seguridad del cliente. Esto es, requieren que cada programa cliente usado actué correctamente de acuerdo con las indicaciones de su usuario.

Es posible ganar un alto nivel de confianza en el cliente de la comisión o el cliente del auditor mediante el uso de firmas digitales. Diversos grupos independientes pueden verificar manualmente el código de estos programas y añadir firmas digitales a aquellas versiones que certifiquen como correctas. La publicación de todos los componentes de PloneVote como software libre

facilita tal verificación. El usuario de estos clientes puede entonces comprobar que su versión del código fue firmada por al menos una entidad en la cual confía[16]. Por supuesto, cualquier usuario puede auditar el código personalmente, si posee los conocimientos técnicos necesarios para ello.

Es más difícil comprobar el correcto funcionamiento del cliente web del elector, ya que dicho cliente es provisto por el propio servidor de PloneVote[17] durante la elección. Más aun, el servidor de PloneVote podría entregar distintas versiones, correctas o incorrectas, del cliente, a distintos usuarios.

Que el cliente web del elector funciona correctamente puede ser asegurado, sin lugar a error alguno, si cada elector verifica el código de dicho cliente de forma manual y experta. En la práctica, tal escenario es improbable. Sin embargo, es posible, con alta probabilidad, descubrir a un servidor que provee un cliente web malicioso o incorrecto, mediante la auditoría del proceso de captura (ver sección 5.2.3.2).

Si suponemos que el servidor provee un cliente web del elector incorrecto o malicioso tan solo 10 de un total de 100 veces que dicho cliente es invocado, y 10 de las invocaciones del cliente web del elector son realizadas por un auditor usando el cliente del auditor para verificarlo, entonces la probabilidad de que alguna de esas verificaciones descubra el cliente malicioso es

$$p = 1 - \frac{90 \cdot 89 \cdot 88 \cdot 87 \cdot 86 \cdot 85 \cdot 84 \cdot 83 \cdot 82 \cdot 81}{100 \cdot 99 \cdot 98 \cdot 97 \cdot 96 \cdot 95 \cdot 94 \cdot 93 \cdot 92 \cdot 91} \approx 0{,}67.$$

Esto es, un servidor malicioso que busque alterar el 10 % de los votos, elegidos al azar, será detectado con probabilidad mayor al 66 % con tan solo un 10 % de auditorías. También para 100 invocaciones del cliente web, 20 auditorías serían capaces de detectar con probabilidad mayor a 68 % un 5 % de alteraciones por parte del servidor.

La probabilidad de detectar un ataque de este tipo se incrementa con el número de veces que es invocado el cliente web del elector, si la proporción de estas invocaciones que son auditorías se mantiene constante. Para 1000 invocaciones totales del cliente, 5 % de auditorías (50) detectan un 5 % de alteraciones con 92 % de probabilidad, y un 1,5 % de alteraciones con 54 % de probabilidad.

Así pues, un servidor malicioso tratando de alterar los resultados de la elección, mediante la entrega de un cliente malicioso a una cierta proporción de los electores al azar, será fácilmente detectado con un esfuerzo relativamente pequeño por parte de los auditores y muy alta probabilidad. El alto riesgo de ser descubierto, y de que la alteración sea rastreada al administrador o atacante responsable, debería ayudar a mantener al servidor honesto.

Proveer un cliente del elector como aplicación de escritorio, sujeto a la posibilidad de ser firmado digitalmente y comprobado usando las mismas

[16] Múltiples sistemas operativos actuales (Windows [SignTool], Ubuntu Linux [DPKG-SIG], etc) permiten verificar automáticamente las firmas digitales de los programas que son instalados en ellos.

[17] El cual, como ya vimos, no necesariamente podemos considerar confiable.

técnicas mencionadas para el cliente de la comisión y del auditor, es una alternativa a contemplarse para versiones futuras de PloneVote.

5.3. PloneVoteCryptoLib

En las secciones 2.3, 3.3 y 4.3, describimos la interfaz externa de la biblioteca PloneVoteCryptoLib, la cual implementa los primitivos criptográficos discutidos a lo largo este documento. No repetiremos tal información en esta sección, presentando brevemente, en cambio, las contribuciones de esta biblioteca al sistema PloneVote.

PloneVoteCryptoLib soporta la mayor parte de la funcionalidad criptográfica requerida por PloneVote, incluyendo los conceptos de redes de mezcla y cifrado de umbral, de los cuales no conocemos ninguna otra implementación adecuada para nuestro sistema. En ese sentido, este componente forma el núcleo de seguridad de nuestro sistema para elecciones en línea.

Existen tan solo dos piezas notables de funcionalidad criptográfica utilizada en PloneVote que no son parte de PloneVoteCryptoLib. La primera es la función de hash segura SHA-256, implementada directamente por pycrypto ([pycrypto]). La segunda es una implementación de cifrado ElGamal simple (Alg. 2.4) en JavaScript, que forma parte del cliente web del elector.

PloneVoteCryptoLib fue desarrollada en python, sin dependencias de Zope o Plone, permitiendo su invocación tanto desde el servidor de PloneVote como desde el cliente de la comisión y del auditor. La biblioteca presenta una interfaz natural a los componentes de PloneVote, al haber sido diseñada como parte del sistema.

Las interacciones esperadas del sistema final con los servicios de PloneVoteCryptoLib incluyen las siguientes:

- Durante la fase de configuración de la elección, el cliente de la comisión utilizará PloneVoteCryptoLib para generar el par de llaves de ElGamal estándar usadas para establecer un canal seguro con el resto de los miembros de la comisión.
- Durante la fase de configuración, el cliente de la comisión utilizará la funcionalidad de **plonevotecryptolib.Threshold** para realizar ambas rondas del proceso de establecimiento del esquema de umbral para la elección, generando la llave pública general de la elección y las llaves privadas de umbral de cada miembro.
- El cliente de la comisión también hará uso de PloneVoteCryptoLib para obtener la huella digital de la llave pública generada, la cual debe ser comparada con la publicada por el servidor.
- El servidor de PloneVote utilizará PloneVoteCryptoLib, y los formatos definidos por esta biblioteca, para almacenar la llave pública general de la elección y obtener su huella digital.

- Durante la fase de votación, el cliente del auditor utilizará PloneVoteCryptoLib para comprobar que la huella digital de la llave pública usada por el cliente web del elector coincida con la publicada por la comisión de la elección.
- Durante la fase de conteo, el cliente de la comisión utilizará **plonevotecryptolib.Mixnet** para realizar mezclas verificables de la colección de votos. El servidor usará este mismo paquete para verificar dichas mezclas antes de aceptarlas.
- Durante la fase de conteo, el cliente de la comisión utilizará **plonevotecryptolib.Threshold** para generar los descifrados parciales de los votos dentro de la última colección mezclada, junto con sus pruebas de descifrado parcial correspondientes.
- El cliente del auditor utiliza también PloneVoteCryptoLib para comprobar el proceso de conteo completo durante la fase de auditoría, delegando a **plonevotecryptolib.Mixnet** y **plonevotecryptolib.Threshold** la verificación de las pruebas de mezcla y descifrado parcial, respectivamente.
- Tanto el servidor de PloneVote como el cliente del auditor requieren combinar los descifrados parciales para obtener el texto claro de los votos de la última colección mezclada. Este proceso es realizado mediante la clase *ThresholdDecryptionCombinator* dentro de **plonevotecryptolib.Threshold**.

Aunque el sistema PloneVote completo es un proyecto aún en desarrollo, y su diseño consta de múltiples componentes complejos, consideramos que PloneVoteCryptoLib es la pieza individual más importante de tal sistema, y representa un paso significativo hacia la implementación de elecciones en línea seguras y verificables sobre el manejador de contenidos Plone.

Durante nuestra implementación de la biblioteca, tuvimos particular cuidado en construir un componente bien estructurado y documentado, incluyendo una suite de pruebas que verifique su correcto comportamiento, así como especificaciones de todas las clases, propiedades, argumentos y métodos considerados públicos. En aquellos métodos que implementan algoritmos criptográficos o numéricos, tuvimos especial cuidado en comentar los distintos pasos realizados por el algoritmo y proveer enlaces a referencias que expliquen sus propiedades no triviales.

Capítulo 6

Conclusiones

Comenzamos el presente documento con una exposición del problema elecciones en línea, haciendo particular énfasis en las propiedades de seguridad que deben ser cumplidas por un esquema de votación ideal. Notamos también que, para que un sistema de elecciones en línea sea suficientemente confiable para el electorado, al menos las propiedades de correctitud y privacidad deberían ser garantizadas sin necesidad de contar con un agente confiable, incluyendo al propio servidor que coordina la elección.

Durante la mayor parte del trabajo, describimos los distintos primitivos criptográficos que requerimos para nuestra construcción de un sistema de elecciones en línea seguro, de acuerdo con nuestro análisis inicial. En cada caso, partimos del comportamiento deseado de tales primitivos, para pasar posteriormente a describir los detalles de la teoría que permite su realización y, finalmente, a explorar nuestra implementación de estos primitivos dentro de PloneVoteCryptoLib.

Finalmente, en el capítulo 5, dimos el diseño general y el protocolo seguido por PloneVote, nuestra propuesta de sistema para elecciones en linea. Mostramos como, utilizando los primitivos y funciones criptográficas implementados en PloneVoteCryptoLib y descritas en los capítulos anteriores, PloneVote debería ser capaz de satisfacer, bajo supuestos razonables, las propiedades de seguridad que nos interesan para elecciones intra-institucionales.

PloneVote permite a cada elector emitir su voto desde cualquier navegador web moderno y tener una garantía verificable de que dicho voto fue capturado correctamente en la colección original de votos en el servidor, mediante el recibo de voto. El elector puede confiar en que este recibo fue generado de forma correcta y en que su voto sólo es guardado en forma cifrada, puesto que el código del cliente web que realiza estas acciones se ejecuta dentro de su equipo de cómputo personal, bajo el control del elector.

Una vez capturados los votos, éstos son procesados mediante una red de mezcla en que participan los miembros de la comisión de la elección. La red de mezcla anonimiza los votos, haciendo imposible asociar a un voto

con su respectivo elector, siempre que exista al menos un miembro honesto de la comisión participando en la mezcla. Usamos pruebas de mezcla en conocimiento cero, para garantizar que los votos al final de todos los procesos de mezcla corresponden al conjunto original de votos capturados.

Tras ser mezclados, los votos son descifrados y contados de forma públicamente verificable. Es necesaria la participación de una mayoría de la comisión de la elección para realizar el proceso de descifrado, lo cual garantiza que, si al menos la mitad de la comisión es honesta, los votos sólo sean descifrados una vez que éstos hayan sido anonimizados mediante la red de mezcla.

Un auditor honesto es capaz verificar la correctitud del conteo de votos en una elección en PloneVote de forma esencialmente incondicional. Además, PloneVote garantiza al elector la privacidad de su voto, siempre que no exista colusión de la mayoría de la comisión de la elección para violar dicha privacidad. Otras propiedades de seguridad deseables dependen directamente de las dos anteriores (e.g. verificabilidad y justicia). Finalmente, algunas propiedades de seguridad, consideradas de menor prioridad, son provistas por PloneVote sólo cuando el servidor se comporta de forma correcta (e.g. robustez).

Una limitación de nuestro sistema es la falta de protección contra coerción. Sin embargo, consideramos improbable que éste sea un problema significativo en el escenario de elecciones intra-institucionales, el cual es el que buscamos soportar mediante PloneVote.

Consideramos que el sistema PloneVote, tal como es descrito en el capítulo 5, provee las garantías de seguridad, capacidades y facilidad de uso adecuadas para realizar elecciones internas de universidades, empresas y comunidades en línea. En particular, proponemos su implementación y uso para las elecciones internas del Instituto de Matemáticas de la Universidad Nacional Autónoma de México.

La biblioteca PloneVoteCryptoLib, desarrollada como parte del trabajo aquí descrito, da una implementación fácil de usar de los complejos primitivos criptográficos necesarios para soportar el protocolo de elecciones de PloneVote. Asimismo, el presente documento muestra cómo dichos primitivos son realizados, a nivel matemático, así como la forma en que proveen las garantías requeridas de ellos por el protocolo y sistema de elecciones completo. Cuando dichas garantías dependen de supuestos particulares, tales como la dificultad del problema de Diffie-Hellman o la resistencia ante pre-imagen de SHA-256, hemos buscado identificar dichos supuestos de forma explícita.

Es nuestra esperanza que el desarrollo del sistema PloneVote continúe después de este trabajo, utilizando los servicios provistos por PloneVoteCryptoLib, para soportar un protocolo seguro para elecciones intra-institucionales. Consideramos que, con este trabajo, estamos entregando una base sólida, tanto en teoría como en implementación, que facilitará que tal esfuerzo sea exitoso.

Bibliografía

[MOV1996] **A. Menezes, P. van Oorschot and S. Vanstone**. *Handbook of Applied Cryptography*, CRC Press, 1996.

[Reyzin2004] **Leonid Reyzin**. *Fundamentals Cryptography*, Course Lecture Notes, Boston University, 2001-2004, http://www.cs.bu.edu/~reyzin/teaching/cryptonotes/

[Shannon1949] **Claude E. Shannon**. *Communication theory of secrecy systems*, Bell System Technical Journal, 28(4):656-715, October 1949.

[Kerckhoffs1883] **Auguste Kerckhoffs**, *La cryptographie militaire*, Journal des sciences militaires, vol. IX, pp. 5–83, Jan. 1883, pp. 161–191, Feb. 1883, http://www.petitcolas.net/fabien/kerckhoffs/ (artículo en francés, principios traducidos al inglés)

[Schneier2002] **Bruce Schneier**, *Secrecy, Security, and Obscurity*, Crypto-Gram Newsletter, May 15, 2002, http://www.schneier.com/crypto-gram-0205.html#1

[TrueCrypt] TrueCrypt, http://www.truecrypt.org/docs/

[dm-crypt] dm-crypt Linux Kernel Module, http://www.mjmwired.net/kernel/Documentation/device-mapper/dm-crypt.txt (also found as part of the Linux Kernel Documentation)

[BitLocker] **Microsoft Corporation**, *BitLocker Drive Encryption Overview*, Microsoft Technet Library, http://technet.microsoft.com/en-us/library/cc732774.aspx

[NIST1999] **National Institute of Standards and Technology**, *Data Encryption Standard (DES)*, Federal Information Processing Standards Publication 46, revision 3 (US FIPS 46-3), October 25, 1999.

[DR1998] **J. Daemen and V. Rijmen**, *AES Proposal: Rijndael*, NIST AES Proposal, Jun 1998.

[NIST2001] **National Institute of Standards and Technology**, *Announcing the Advanced Encryption Standard (AES)*, Federal Information Processing Standards Publication 197 (US FIPS 197), November 26, 2001.

[Schneier1994] **Bruce Schneier**, *Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)*, Fast Software Encryption, Cambridge Security Workshop Proceedings (December 1993), Springer-Verlag, 1994, pp. 191-204.

[S et. al. 1998] **B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, N. Ferguson**, *Twofish: A 128-Bit Block Cipher*, 15 June 1998, http://www.schneier.com/paper-twofish-paper.html

[ABK1998] **Ross Anderson, Eli Biham, Lars Knudsen**, *Serpent: A Proposal for the Advanced Encryption Standard*, http://www.cl.cam.ac.uk/~rja14/Papers/serpent.pdf

[RSA1978] **Rivest, R.; A. Shamir; L. Adleman**, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the ACM 21 (2): 120–126, Feb. 1978.

[ElGamal1985] **Taher ElGamal**, *A public key cryptosystem and a signature scheme based on discrete logarithms*, Advances in cryptology: Proceedings of CRYPTO 84. Lecture Notes in Computer Science. 196. Santa Barbara, California, United States: Springer-Verlag. pp. 10–18, 1985

[Paillier1999] **Pascal Paillier**, *Public-Key Cryptosystems Based on Composite Degree Residuosity Classes*, EUROCRYPT 1999, pp223-238.

[GM1984] **S. Goldwasser and S. Micali**, *Probabilistic encryption*, Journal of Computer and System Sciences, 28:270-299, 1984.

[Waters2009] **Brent Waters**, *Introduction to Cryptography*, Course Lecture Notes, University of Texas at Austin, 2009, http://www.cs.utexas.edu/~sarak/cs388h/

[FSF1999] **The Free Software Foundation**, *The GNU Privacy Handbook*, 1999, http://www.gnupg.org/documentation/guides.en.html

[IETF2008] **Internet Engineering Task Force**, *The Transport Layer Security (TLS) Protocol Version 1.2*, RFC 5246, August 2008, http://tools.ietf.org/html/rfc5246

[BGB2004] **Nikita Borisov, Ian Goldberg, Eric Brewer**, *Off-the-Record Communication, or, Why Not To Use PGP*, Workshop on Privacy in the Electronic Society, October 28, 2004

[IETF1998] **Internet Engineering Task Force**, *OpenPGP Message Format*, RFC 2440, November 1998, http://www.ietf.org/rfc/rfc2440.txt

[Benaloh2006] **Josh Benaloh**, *Simple Verifiable Elections*, EVT'06 Proceedings of the USENIX/Accurate Electronic Voting Technology Workshop 2006 on Electronic Voting Technology Workshop, USENIX Association, June 14, 2006.

[CP2001] **Crandall, R. and Pomerance, C.**, *Prime Numbers; a computational perspective*, Springer Verlag, New York 2001.

[Herstein1975] **I. N. Herstein**, *Topics in Algebra*, John Wiley & Sons, 2nd Edition, Jan 1975.

[Studholme2002] **Chris Studholme**, *The discrete log problem*, June 21, 2002, http://www.cs.toronto.edu/~cvs/dlog/

[GPG2008] 'gpg –gen-key' inline instructions, *gpg (GnuPG) 1.4.10*, Free Software Foundation, 2008

[NIST2007] **National Institute of Standards and Technology**, *Recommendation for Key Management – Part 1: General (Revised)*, NIST Special Publication 800-57, March 2007

[pycrypto] PyCrypto - The Python Cryptography Toolkit, http://www.dlitz.net/software/pycrypto/

[GHJV1995] **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**, *Design Patters: Elements of Reusable Object-Oriented Design*, Addison-Wesley, 1995

[Shamir1979] **Adi Shamir**, *How to Share a Secret*, Communications of the ACM, November 1979

[Knuth1969] **Knuth, D**, *The Art o f Computer Programming, Vol. 2: Seminumerical Algorithms*, Addison-Wesley, 1969.

[CGS1997] **Ronald Cramer, Rosario Gennaro, and Berry Schoenmakers**, *A secure and optimally efficient multi-authority election scheme*, EUROCRYPT, volume 1233 of Lecture Notes in Computer Science, pages 103–118. Springer, 1997.

[Pedersen1991] **Torben P. Pedersen**, *A threshold cryptosystem without a trusted party (extended abstract)*, EUROCRYPT, volume 547 of Lecture Notes in Computer Science, pages 522–526. Springer, 1991.

[GMR1985] **Shafi Goldwasser, Silvio Micali, and Charles Rackoff**, *The knowledge complexity of interactive proof-systems (extended abstract)*, In STOC, pages 291–304. ACM, 1985.

[Goldreich2002] **Oded Goldreich**, *Zero-knowledge twenty years after its invention*, Electronic Colloquium on Computational Complexity (ECCC), (063), 2002.

[CP1992] **David Chaum and Torben P. Pedersen**, *Wallet databases with observers*, CRYPTO '92 Proceedings of the 12th Annual International Cryptology, 1992

[FS1986] **Amos Fiat and Adi Shamir**, *How to prove yourself: Practical solutions to identification and signature problems*, In Andrew M. Odlyzko, editor, CRYPTO, volume 263 of Lecture Notes in Computer Science, pages 186– 194. Springer, 1986.

[IETF2006] **Internet Engineering Task Force**, *US Secure Hash Algorithms (SHA and HMAC-SHA)*, RFC 4634, July 2006, http://tools.ietf.org/html/rfc4634

[Chaum1981] **D. Chaum**, *Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms*, Communications of the ACM, 24(2):84-88, February 1981.

[PIK1994] **C. Park, K. Itoh and K. Kurosawa**, *Efficient Anonymous Channel and All/Nothing Election Scheme*, In EUROCRYPT '93, volume 765 of LNCS, pages 248-259, Springer-Verlag, 1994

[Plone] Plone CMS, http://plone.org/

[Tidwell2006] **Jenifer Tidwell**, *Designing Interfaces: Patterns for Effective Interaction Design*, O'Reilly Media, 2006

[Zapata2008] **Alexander Zapata Lenis**, *Implementación de un sistema de votación electrónica como un producto sobre la plataforma Plone*, Tesis de Maestría, Universidad Nacional Autónoma de México, 2008

[Adida2008] **Ben Adida**, *Helios: Web-based Open-Audit Voting*, Usenix Security, 2008

[Gentry2009] **Craig Gentry**, *Fully Homomorphic Encryption Using Ideal Lattices*, STOC '09 Proceedings of the 41st annual ACM symposium on Theory of computing, 2009

[DPKG-SIG] **A. Barth and M. Brockschmidt**, *dpkg-sig man page*, Ubuntu Manual (http://manpages.ubuntu.com/manpages/natty/man7/dpkg-sig.7.html)

[SignTool] *SignTool.exe (Sign Tool)*, MicroSoft Developer Network (MSDN) Library, April 2011 (http://msdn.microsoft.com/en-us/library/8s9b9yaz.aspx)

[Rotman2002] **Joseph Rotman**, *Advanced Modern Algebra*, Prentice Hall, 2002