UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE ELECTRÓNICA
VALPARAÍSO - CHILE

"SISTEMA DE I-VOTING BASADO EN TECNOLOGÍA "BLOCKCHAIN" Y ANONIMATO MEDIANTE PRUEBAS DE CERO CONOCIMIENTO"

MATIAS DÍAZ

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL ELECTRÓNICO

PROFESOR GUIA: **JOSÉ MARTÍNEZ**

PROFESOR CORREFERENTE: **MAURICIO ARAYA**

A mis padres.

A todo lo libre, noble, puro y virtuoso.

Resumen

Recientemente tecnologías disruptivas en área de la informática y criptografía han dado una nueva perspectiva de soluciones a problemas presentes en la sociedad como lo son los sistemas de votaciones. Las tecnologías de ”blockchain” y pruebas de cero conocimiento ofrecen seguridad, transparencia, anonimato y fiabilidad como ninguna otra tecnología lo ha hecho hasta el momento.

En la presente memoria se propone un sistema de voto electrónico que cumpla con las características de ser completo, fiable, transparente, verificable, seguro y anónimo. El sistema propuesto está basado en las tecnologías ya mencionadas además de otras que ayuden al íntegro funcionamiento de esta.

Esta memoria detalla la creación de un piloto que demuestra que es técnicamente viable la creación del sistema de voto descrito. Además se estudiarán formas de hacer más eficiente y rápido el sistema.

Temas relacionados: sistema de voto electrónico, ”blockchain”, pruebas de cero conocimiento.

Abstract

Recently, disruptive technologies in informatics and cryptography brings a new perspective to solutions of problem present in society, like voting systems. Technologies like blockchain and zero knowledge proofs, brings security, transparency, anonymity and reliability as no other technology has done it till now.

The present work proposes an electronic voting system that accomplishes completeness, reliability, transparency, verificability, security and anonymity. The system will be based on the technologies mentioned early and others needed for it rigth functionality.

This work contemplates the creation of a pilot that demonstrates that, a system like this, is technically possible. Futhermore, other forms to make the system more efficient and fast will be studied.

Keywords: electronic voting, blockchain, zero knowledge proofs.

Glosario

API: *Application Programming Interface* Es una capa de software que sirve de interfaz común entre dos aplicaciones para que una tenga acceso a las funciones de la otra.

Ataques DDoS: Ataque de denegación de servicios distribuido por sus siglas en inglés. Consiste en la interrupción del servicio de un sistema mediante el colapso de la red de la víctima producto de un alto tráfico proveniente de varios atacantes.

Booleano: Tipo de dato que representa valores lógicos (verdadero o falso).

Contenedor Docker: Instancia de un sistema aislado que aprovecha el kernel del sistema que lo alberga. Docker es el servicio que provee esta funcionalidad www.docker.com. Funciona similar a una máquina virtual pero mucho más ligera.

Certificado Digital: Documento digital que permite certificar la identidad de una entidad mediante la pertenencia de una llave pública.

Criptomoneda: Moneda digital cuya existencia está definida en un protocolo que utiliza algún método criptográfico para asegurar su correcto funcionamiento. En esta memoria se hablarán solo de aquellas que están definidas en un sistema de cadena de bloques.

Endpoint: En una API el ”endpoint” es una rutina que da acceso a una funcionalidad específica de la API. Es el destino final de un requerimiento.

Estado ”Halt”: Estado final al que llega un proceso en una máquina de Turing.

Firma Digital: Objeto criptográfico que permite comprobar el origen y la integridad de un mensaje. En esquemas asimétricos es generada mediante una llave privada y comprobada por su llave pública correspondiente.

Función Hash: función que convierte una entrada de tamaño aleatorio a un número de tamaño fijo al pasar por un algoritmo tal que no exista forma práctica de deducir la entrada al conocer la salida.

Git: Es un protocolo de manejo de versiones de un recurso. Los recursos quedan depositados en un repositorio en donde están alojados el recursos y copias de sus modificaciones históricas que pueden ser accedidos por 1 o más entidades.

Http: *HyperText Transfer Protocol*, es un protocolo de comunicación de red en la capa de aplicaciones que se utiliza para el traspaso de páginas web, funciona a través de requerimientos que tienen tipos específicos de acceso o modificación de recursos.
JSON: *JavaScript Object Notation* es un formato de serialización pensado en descrbir objetos del lenguaje JavaScript. Actualmente es el uno de los formatos más usados para serializar estructuras de distintos lenguajes en distintas aplicaciones.
Lenguaje Ensamblador: Lenguaje de programación basado en ”Opcodes” o instrucciones cortas interpretadas directamente por una máquina.
Lenguaje Turing-completo: Lenguaje de programación que permite codificar cualquier tipo de problema que puede resolverse en una máquina de Turing universal.
Máquina de estado: Modelo de comportamiento que describe el estado interno de un sistema el cual va siendo modificado según las entradas del sistema y el estado actual en que se encuentra.
LDAP: Protocolo de acceso a un sistema de fichero basado en las credenciales del dueño. En linux tiene un servicio llamado OpenLDAP.
Llave privada: En encriptación asimétrica la llave privada es un dato que sólo conoce su dueño y sirve para cifrar, descifrar y generar firmas digitales en esquemas asimétricos en conjunto con su par público.
Llave pública: En encriptación asimétrica la llave pública es un dato complementario a la llave privada que puede ser conocido por cualquier entidad sin comprometer el esquema de encriptación, esta sirve para decifrar, cifrar y verificar firmas digitales creadas por su par privado.
Máquina virtual: Emulador de un sistema dentro de otro. Tiene su propio sistema operativo, sistema de ficheros,usuarios y recursos. Una máquina virtual está completamente aislada de su huésped y consume sus recursos.

Moneda Digital: Activo monetario que existe sólo de forma digital y que puede utilizarse como medio de intercambio de bienes y servicios. **MSP**: *Membership Service Provider* Servicio que maneja las entidades en Hyperledger Fabric explicado en la

sección3.4.4

Nodo: Participante individual que, al conectarse con otros nodos, conforman una red en conjunto.

OAuth: Protocolo que permite el flujo de credenciales para el uso de APIs en paginas web.

Opcode: Instrucción informática parte de un lenguaje máquina o ensamblador que realiza una operación.

Proxy: Un servidor Proxy es un servicio en la red que actúa en nombre de otro ”host”. Es usado como antesala de algunos servidores web para controlar y manejar de mejor forma el tráfico de entrada.

RNG: Generador de números aleatorios por su siglas en inglés.

Query: Consulta específica a una base de datos.

Round robin: Método de selección de miembros de un grupo que funciona de forma equitativa y según el ordenamiento mismo del grupo, comenzando desde el primer elemento hasta el último según dicho orden.

Serialización: Procedimiento para transformar una estructura de dato en un formato compacto que pueda ser transportado por una red como un arreglo de bytes.

Script: Programa informático que consiste en una serie de instrucciones generalmente simples.

TLS: *Transport Layer Security*: Protocolo para la encriptación de canales de comunicación en la red.

TTY:*teletypewriter*: En Linux, una tty es una terminal de comandos en donde un usuario con una sesión abierta puede interactuar mediante comandos con el sistema operativo.

Yaml: Es un tipo de formato de serialización como JSON o XML.

ZKP: *Zero Knwoledge Proof*, pruebas criptograficas del conocimiento de algo sin revelar su valor como se detalla en la sección 5.2

CONTENTS

1. INTRODUCCIÓN

A medida que las poblaciones crecen y las necesidades se vuelven más complejas, tanto para individuos como para las comunidades, nuevas técnicas de toma de decisiones deben ser investigadas. En el caso de los sistemas de votación, cada vez más se busca reemplazar el sistema tradicional de papeleta y urnas por sistemas electrónicos que ayuden a los países a crear servicios más eficientes, efectivos y amigables para los votantes [47].

Los sistemas remotos de votación existen desde la antigüedad. Sin embargo, solo en las últimas décadas se ha establecido una especia de carrera entre los estados para crear el sistema de voto electrónico remoto (REV) definitivo [48]. En esta carrera, se ha evidenciado las principales falencias y amenazas de los sistemas REV: riesgo de intervención en el hardware y software del sistema, fallas e intervención de los servidores de conteo, vulnerabilidades en los sistemas criptográficos que protegen la identidad de los votantes, y vulnerabilidad y ataques tipo ”man in the middle” en los canales de conexión del sistemas, entre otros.

Sin embargo las tecnologías han cambiado y nuevos sistemas de confianzas y técnicas criptográficas han sido desarrolladas en los últimos años. La llegada de tecnologías como ”blockchain” [1] y las pruebas de cero conocimiento [27] abren nuevas posibilidades de sistemas de votación más robustos y confiables.

En la presente memoria se plantea la arquitectura de un sistema de votación electrónico mediante el uso de redes de computadoras y nuevas tecnologías informáticas, como lo es Hyperledger Fabric - un sistema que hace uso del ”blockchain” en su arquitectura - y ZK-SNARK -un esquema de pruebas de cero conocimiento. En este trabajo, se describirán estas tecnologías, se verán las ventajas, su limitaciones y de qué man-

era mejoran a los sistemas de votación electrónicos presentes en el estado del arte. Posteriormente se describirá el desarrollo del piloto y se presentará el programa de votación, cómo funciona y cómo fue implementado. Finalmente se mostrarán los resultados obtenidos y se compararán con los objetivos planteados.

1.1 Estado del Arte

Remote Electronic Voting

Desde hace un tiempo existen sistemas de votación remotos, siendo el sistema postal de votación el registro más antiguo de este, el cual data desde el antiguo Imperio Romano y que, aún en la actualidad, es usado [48]. Posteriormente fueron incorporados servicios como la telefonía y el fax. La llegada del internet gatilló un inmediato intento por hacer sistemas de voto electrónico remoto (REV, por sus siglas en inglés) al principio mediante redes privadas en donde la privacidad del votante no era una prioridad. Desde 1990 comienza una carrera política global para ver qué país implementaría el primer sistema REV. Pero el resultado de las elecciones presidenciales de Estados Unidos del 2001, entre los candidatos G. Bush vs A. Gore, concentró la atención de los estadounidenses, en cuanto a la seguridad e integridad de su sistema de votación [48].

Sistemas REV End to End

Para verificar la seguridad e integridad de los sistemas, debemos considerar el concepto de End-to-End (E2E) en las REV, que consiste en que el votante es capaz de verificar que su voto fue *contado con su valor correcto* [49].

En lo que concuerdan estos sistemas REV-E2E es en emplear sistemas criptográficos para [50]:

- Proteger la identidad del votante (privacidad).

- Asegurar que los votos fueron contados con el valor que el votante otorgó al voto (integridad).
- Cifrar los resultados y sólo revelarlos al momento de contar los votos (justicia).
- Poder cambiar el valor del voto en el plazo correspondiente (resistencia a la coerción y perdón).
- Que sólo los votantes inscritos en el padrón electoral puedan emitir votos válidos (elegibilidad).

Muchos de estos sistemas agregan además un tablón de anuncios público en la web (WBB por sus siglas en inglés) en donde se publican los votos emitidos encriptados. Así los votantes pueden verificar, mediante una clave, si su voto fue contado como corresponde [49], con ello transparentando un poco la ejecución del sistema. Sin embargo varios de estos sistemas REV-E2E que consisten en una estructura cliente-servidor presentan varios problemas de seguridad del tipo server-side y client-side [51] además de malas prácticas de mantención a los servidores y ejecución que generan inseguridades que pueden ser explotadas mediante ingeniería social [52].

1.1.1 Casos de interés

El primer sistema revisado es el I-voting, el sistema de votación nacional de Estonia [52][47]. Este sistema hace uso de una tarjeta nacional de identificación en donde los votantes (y los ciudadanos en general) tienen sus pares de llave privada y llave pública del protocolo RSA [52]. Mediante estas llaves los votantes se autentican en el sistema, mediante un servicio LDAP con conexión TLS [52]. Una vez autenticado el votante procede a seleccionar a su candidato generando un voto. Este voto es cifrado mediante una firma OAEP generada por las credenciales de su tarjeta y un número "r" aleatorio[52]. Una vez cifrado y firmado el voto, este es subido a la red del sistema y le es devuelto al votante un código QR con el cual él puede verificar que su voto fue contado de manera adecuada.

La infraestructura del sistema cuenta con 4 servidores [52]:

- Vote forwarding server (VFS): Es el único servidor con acceso público, verifica la autenticación de los clientes y sirve como interfaz entre el cliente y el backend del sistema.
- Vote storage server (VSS): Es aquel servidor que guarda los votos firmados y cifrados.
- Log server: Es el servidor que guarda y administra los logs de los servidores durante todo el proceso.
- Vote counting server (VCS): Es el servidor que cuenta los votos y es utilizado sólo cuando el proceso termina. Los votos son cargados a este servidor mediante DVD, por lo cual este servidor nunca es conectado a internet.

La app (aplicación móvil) puede ser descargada desde su página oficial y sirve tanto para emitir votos como para verificarlos mediante el código QR. Además, el sistema permite sobrescribir los votos, con votos posteriores, a modo de defensa contra la coerción. Como última instancia, el voto en papel sobrescribe cualquier voto realizado en el sistema online. Durante las elecciones generales del 2013 de Estonia, 23% de los votos fueron emitidos en I-Voting [52].

En el caso de las tecnologías "blockchain", existe el caso documentado del sistema de votación basado en el protocolo de ZCash. ZCash es un protocolo de criptomoneda que permite transacciones en donde se puede ocultar la dirección de procedencia, la dirección de llegada y el monto a transferir, mediante ZK-SNARKs [49]. Esto permite realizar un sistema de votación anónimo utilizando los tipos de cuenta de ZCash y las transacciones anónimas entre ellas. Dicho sistema se detalla a continuación, primero se debe entender los tipos de transacción entre las cuentas. Zcash tiene 2 tipos de cuentas: t-addresses que son direcciones desprotegidas y z-addresses que son direcciones totalmente protegidas. Una transacción completamente anónima se realiza desde una z-address a otra. Una transacción normal se hace entre dos t-address, una

transacción de ocultamiento de fondos se hace desde una t-address a una z-address y una transacción de revelación de fondos se hace desde una z-address a una t-address.

El sistema en total puede ser separado en 4 etapas [49]:

- Registro: El votante se registra y autentica mediante su dirección email en una página web usando *Challenge-Handshake Authentication Protocol (CHAP)*, en el cual los certificados del votante quedan registrados en una base de datos.

- Invitación: El votante recibe una invitación (un link), la cual lo lleva a una cartola de votación online en donde puede realizar su voto

- Votación: La votación se hace mediante transacciones de ZEC (la criptomoneda de ZCash). Se realizan dos transacciones: una normal (entre t-addresses), en donde el votante manifiesta que entregó su voto, y otra oculta (entre z-adresses), en donde el votante entrega el valor de su voto en forma privada. La t-address de llegada en la primera transacción es una address pública auditable que permite transparentar a los votantes, mientras que la z-address de llegada corresponde a la dirección del candidato por el cual el usuario quiere votar. Posteriormente se comparan la cantidad de transacciones de la t-address con la de todas las z-address. Se considera válido si ambas cantidades coinciden. El problema de las transacción entre z-addresses es que es tan anónimo que el votante no puede verificar si su voto fue contado de forma efectiva (no se cumple E2E). Existe una variante en la cual la transacción de voto se hace desde una z-address hacia una t-address (una transacción de revelación), esto permite E2E, pero además permite que los votos puedan ser contados en tiempo real, incumpliendo con la propiedad de Fairness (justicia).

- Conteo de votos: Es la etapa final en donde los candidatos hacen una transferencia desde sus z-address (o t-address si se elige la segunda variante) hacia una t-address que sirve para contar los votos.

1.2 Objetivos

Con estos antecedentes se pueden establecer los objetivos a cumplir en esta memoria. Se desea implementar un sistema de ”Remote Electronic Voting” (REV) que permita a los votante elegir a su preferente entre una lista de candidatos. Esta elección es emitida y publicada para su posterior conteo, si el votante está habilitado para ello. El votante sólo podrá emitir un voto válido durante el proceso. Además, este proceso debe cumplir con las siguientes propiedades:

- Integridad de los datos: Los datos publicados en el sistema se mantienen como fueron intencionalmente construidos y su flujo obedece estrictamente a un protocolo establecido y conocido.
- Disponibilidad del servicio: El sistema mantiene todas sus funcionalidades accesibles durante el proceso.
- Escalabilidad: El servicio mantiene sus propiedades y funcionalidades independiente de la cantidad de participantes (tanto de infraestructura como de clientes) y de a taza de crecimiento de estos.
- Verificación E2E: Los votantes pueden verificar que su voto fue debidamente contado.
- Verificación pública: Cualquier entidad (persona, institución), que lo desee, puede verificar que los votos fueron válidamente emitidos y que el resultado final se condice con los votos publicados.
- Privacidad: El voto publicado no podrá vincularse con ningún votante.
- Fairness: El valor de los votos es conocido sólo al final del proceso de votación. No existe un resultado parcial previo al conteo de votos que pueda influenciar, de alguna forma, a los votantes.

- Forgiveness/Resistencia a coerción: Los votantes pueden cambiar el valor de su voto mientras el proceso de votación está activo, de tal manera de proteger su real intención de voto.

- Seguridad: El sistema mantiene todas sus propiedades durante el proceso, independiente de los esfuerzo de un adversario.

Las tecnologías base para este sistema son ”Blockchain” y las ”Zero Knowledge Proofs” (ZKPs). Además, el sistema contempla que los votantes se identifican mediante un rol único y unas credenciales que permitan verificar que es quien dice ser. Junto a ello, los participantes que mantienen la infraestructura, también son identificables y pueden ser auditados.

Esta memoria se organiza en capítulos que describen las tecnologías a utilizar. Posteriormente se detalla como estas tecnologías se aplican en conjunto. En los siguientes capítulos se describe la construcción del piloto, con los ”softwares” involucrados. Finalmente se presentan los resultados, las conclusiones y el trabajo futuro del sistema presentado.

2. ”BLOCKCHAIN”

El 2009 se hace público un articulo [1], el cual describe un sistema de pago en línea llamado ”Bitcoin”. Como se detalla en [1], este sistema consiste en la escritura de transacciones de una moneda digital en una estructura de datos llamada bloque. Esta estructura tiene la cualidad de poseer un identificador único generado a partir del aplicar una función ”Hash” al contenido del bloque, de tal forma que, si su contenido es modificado, ya no se corresponde con su identificador. Además, se incluye en el contenido del bloque el identificador único del bloque anterior, lo que ”encadena” a los bloques como se aprecia en la figura 2.1 (de ahí el término ”blockchain”, cadena de bloques). Esta dinámica de bloques encadenados permite ordenarlos de forma secuencial, por lo que se puede tener un historial del flujo de las transacciones y el balance actualizado de los participantes.

Sin embargo, si un solo nodo mantiene toda la cadena de bloques, nada lo detiene a él o a un atacante de alterar un bloque y re-calcular la cadena nuevamente. Es por eso que, para que el sistema funcione correctamente, se necesita agregar múltiples nodos independientes entre sí, que consensúen una cadena de bloques única.

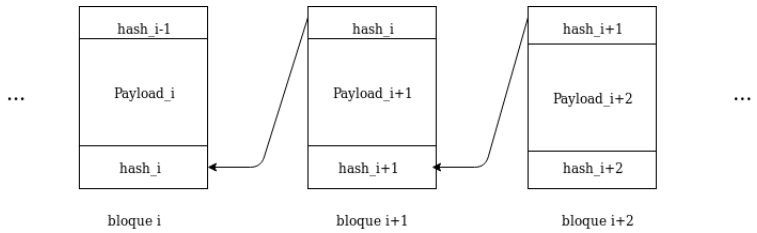


Fig. 2.1: Cadena de bloques, el ”hash” superior equivale al identificador único del bloque anterior y el ”hash” inferior al propio del bloque el que se construye usando el ”payload” y el identificador del bloque anterior.

2.1 Problema de los generales Bizantinos

Si la red acepta cualquier participante puede que el consenso quede comprometido. Este fenómeno llamado ”el problema de los generales Bizantinos” es ampliamente discutido en [2]. Lamport se refiere a un sistema que requiere el consenso entre nodos, el cual debe ser robusto frente al comportamiento errático, e incluso deliberadamente malicioso, de un cierto porcentaje de participantes de la red.

Para que una red de nodos independientes, que implementa una cadena de bloques, funcione correctamente debe implementar un método de consenso tolerante a la falla bizantina. Esto quiere decir que, en una red de M nodos se puede llegar a un consenso honesto si a lo más N nodos tienen un comportamiento errático y/o malicioso, donde esta cantidad N depende del método utilizado.

2.1.1 ”Proof of Work”

Bitcoin utiliza un método de consenso llamado ”Proof of Work” [1], el cual es probabilísticamente tolerante a fallas bizantinas [3]. Este consiste en que, una vez que el nodo encuentra el identificador único del bloque, debe combinarlo con un número, llamado ”nonce”, y someter la combinación a una función ”hash”. El resultado de esta función Hash debe comenzar con una cierta cantidad de ”0”, esta cantidad es llamada ”target”. Si el resultado no cumple con la cantidad de 0, se incrementa el ”nonce” y se intenta de nuevo. Este proceso se lleva a cabo reiteradas veces hasta que se cumpla el ”target” de 0. El siguiente pseudo-código muestra el proceso estándar:

```
t = 4
target = "0000"
nonce = 0
h = H(BlockContent)
while(h[:t] != target){
    nonce++
    h = H(BlockContent+nonce)
}
```

```
return BlockContent+nonce+h
```

Donde $H()$ es una función hash resistente a colisiones.

Una vez encontrado el valor ”target”, tanto el ”nonce” como el nuevo identificador único son agregados al bloque. Luego, el nodo publica el bloque encontrado a toda la red para que los demás nodos lo agreguen a la cadena de bloques, si este resulta ser válido.

Mientras mayor sea el ”target” mayor es el poder computacional necesario para encontrar el valor ”Hash” que cumpla con el protocolo. Con esto se puede calcular las probabilidades de un ataque. Sean:
p: la probabilidad de que un nodo honesto encuentre el siguiente bloque.
q: la probabilidad de que un nodo atacante encuentre el siguiente bloque.

Como p es un nodo dentro de toda la red, y asumiendo que los nodos honestos poseen la mayoría del poder computacional, entonces se puede decir que $p > q$ y por lo tanto, la probabilidad q_k de que el atacante reconstruya la cadena desde un bloque k es $q_k : (q/p)^k$ [1].

Es por esto que ”Proof of Work” es un método probabilísticamente tolerante a fallas bizantinas. En [1] se observa que la probabilidad de reescribir la cadena de bloques es de 0.1 % desde $k = 5$ para $q = 0.1$ (10%) y desde $k = 8$ para $q = 0.15$ (15%).

2.1.2 ”Practical Byzantine Fault Tolerance”

Otro ejemplo de método de consenso es el PBFT (”Practical Byzantine Fault Tolerance) [4]. Este método describe una máquina de estados para cada requerimiento al sistema, que determina el accionar de la red según el estado en que se encuentre de tal modo de evitar una falla bizantina. En el caso de ”blockchain”, la máquina de estado es:

1. **Inicio**

- Un nodo líder es elegido (generalmente mediante ”round robin”)
- El líder recolecta las transacciones desde un banco de transacciones.
- El líder genera el bloque y lo difunde a la red. El líder entra en estado ”pre-preparado”.
- Los validadores (el resto de los nodos) reciben el mensaje del líder y entran al estado ”pre-peparado”

2. **Estado: Pre-Preparado**

    - Los validadores verifican el bloque propuesto y difunden un mensaje de ”preparado”
    - Los validadores esperan $f + 1$ mensajes de ”preparado” válidos y entran al estado de ”preparado”

3. **Preparado**

    - Los validadores difunden un mensaje de ”commit”.
    - Los validadores y el líder esperan $2f + 1$ mensajes de ”commit” y entran al estado ”committed”

4. **Committed**

    - Los validadores agregan los mensajes ”commit” al bloque y entran al estado final.

5. **Final Committed**

    - El bloque es agregado a la cadena y todos los nodos vuelven al estado de inicio.

En este caso f es la cantidad máxima de nodos que pueden ser no-honestos o tener alguna falla [4].

El funcionamiento de este algoritmo se puede observar en la figura 2.2. Para un valor f dado, el número total de nodos es de $3f + 1$. Esto se debe a que el protocolo tiene que ser capaz de vencer dos posibles ataques: el primero ocurre cuando f nodos entregan un mensaje malicioso (un mensaje de aprobación para un bloque inválido, un mensaje de desaprobación para un bloque válido o un mensaje ininteligible o fuera de protocolo). Como la decisión se toma mediante voto mayoritario para lograr el $50\% + 1$, los nodos honestos deben ser $f + 1$ en el peor de los casos, lo que da un total de $f + f + 1 = 2f + 1$ de mensajes que los nodos deben esperar. El segundo ataque corresponde a que los f nodos maliciosos no entreguen mensaje alguno. Como los nodos honestos están esperando $2f + 1$ mensajes, deben existir al menos $2f + 1$ nodos honestos. Es por esto que, con un máximo de f nodos maliciosos y un mínimo de $2f + 1$ nodos honesto, la red debe tener $3f + 1$ nodos en total .

Si f no es una variable de diseño, entonces de un total de N nodos f puede ser hasta $\frac{N-1}{3}$ para que el sistema sea resistente a fallas bizantinas.

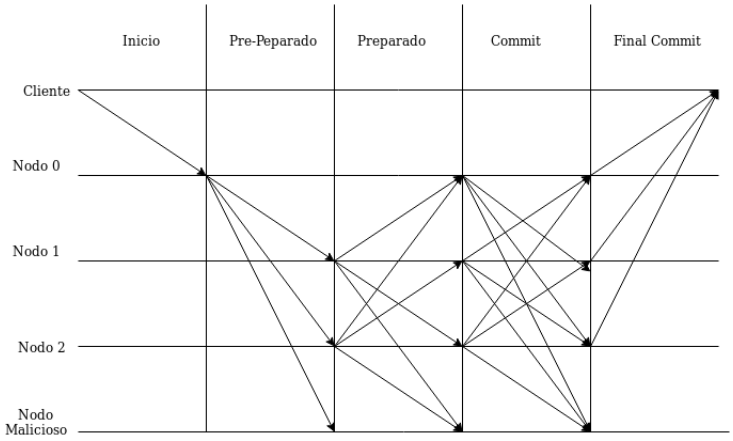


Fig. 2.2: Funcionamiento normal del algoritmo de PBFT [4]. En este caso el nodo 0 es elegido como líder y el último nodo es un nodo que no responde. Aquí se aprecia que $f = 1$, por lo que el total de nodos para que el sistema funcione debe ser de $3f+1 = 4$ y que para hacer ”commit” los nodos esperan $2f + 1 = 3$ mensajes.

2.2 Transacciones y contratos inteligentes

El principal uso de las cadenas de bloque son las criptomonedas. Las más exitosas tienen una propuesta altamente valorada como un método de consenso que las hace más eficiente, seguras o justas. Las transacciones juegan un papel importante en estas propuestas de valor. Una transacción debe ser resistente al fraude, es decir, el método de consenso y el algoritmo de la transacción deben evitar fenómenos como transacciones con cantidades mayores a los fondos, cancelación de la transacción, el doble gasto de una transacción, entre otros eventos de fraude. Pero, además, un esquema de transacción puede agregar valor a la propuesta si esta es más segura, otorga anonimato, permite operaciones complejas u otra característica valorada.

2.2.1 "Pay To Script Hash"

Bitcoin (y similares) ocupan un esquema llamado "pay to script hash" (P2SH) [1][5]. Para hacer transacciones en la red de Bitcoin, los usuarios deben poseer una "billetera" que consiste en un par de llaves privada/pública con cierto formato descrito en el protocolo de Bitcoin. La dirección de esta billetera consiste en el resultado de aplicarle una función "hash" dos veces a la llave pública del usuario. Otros usuarios pueden hacer transferencias de sus Bitcoins a esta dirección.

Como se observa en la figura 2.3, una transacción se compone de "inputs"(entradas) y "outputs"(salidas). Las entradas de una transacción son las salidas de algunas transacciones anteriores. Las salidas van dirigidas a una dirección y tienen un "script" que se compone de instrucciones que deben ser satisfechas para el uso de la salida.

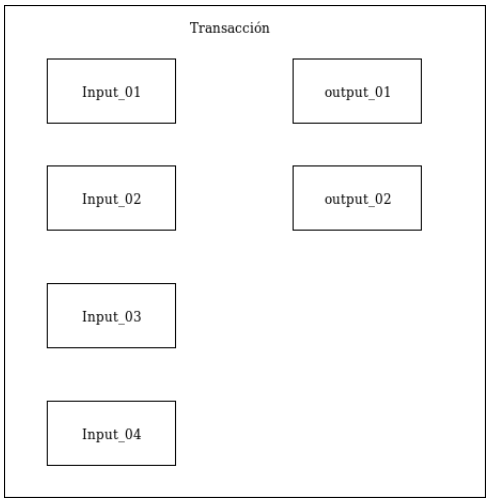


Fig. 2.3: Esquema de una transacción en Bitcoin [1], el nodo que la procesa debe verificar que: 1) La suma de los montos de los ”inputs” es mayor a la suma de los montos de los ”outputs”, 2) Ninguno de los ”inputs” ha sido utilizado en otra transacción. 3) Todos los scripts de los ”inputs” son validados (resultan en el booleano ”true”).

Si alguien quiere hacer uso de una salida de una transacción, usándola como una entrada de su transacción, debe probarle a la red que posee los valores que satisfacen el ”script” de dicha salida. Este es el núcleo de P2SH. El usuario agrega a su transacción la salida de una anterior, además de ciertos valores que, al pasar por el ”script”, retorna un valor de verificación (un booleano ”true”). Los ”scripts” están escritos en ”opcodes” propios de Bitcoin, similares al lenguaje ensamblador que ocupa una pila que se modifica según las instrucciones de los ”opcodes” [5]. El ”script” más común es el ”Standard Transaction to Bitcoin address (pay-to-pubkey-hash)”[5]:

```
1 OP_DUP OP_HASH160 <pubKeyHash> OP_EQUALVERIFY OP_CHECKSIG
```

Este ”script” consume la llave pública y una firma digital generada con el par privado de la llave pública del usuario destinatario, ambos valores son agregados a la pila. ”OP_DUP” duplica en la pila la llave pública. ”OP_HASH160” aplica una función hash dos veces al último elemento de la pila. $< pubKeyHash >$ es agregada a la pila, que consiste en la dirección a la cual va destinada la salida que contiene el ”script”. ”OP_EQUAKVERIFY” verifica que los dos últimos elementos de la pila son idénticos, en este caso $< pubKeyHash >$ y el hash de la llave pública.

”OP_CHECKSIG” verifica la firma digital.

En pseudo código, se vería de la forma:

```
1
2 func Pay2PubKeyHash(PubKey,Sig):
3     if (Hash(PubKey)==pubKeyHash && VerifySign(PubKey,Sig)):
4         return true
5     return false
```

Esto verifica que el usuario entrega la llave pública correspondiente a la dirección destinada de la salida y que dicha llave pública es suya.

2.2.2 Contratos inteligentes

Bitcoin tiene 255 ”opcodes” enumerados [6]. Con estos opcodes se pueden crear ”scripts” más complejos además del ya mencionado ”pay-to-pubkey-hash”. Un usuario no sólo puede transferir Bitcoins sino que puede programar transacciones complejas. Esto da paso a los ”contratos inteligentes”. Que son transacciones programadas con ”opcodes” para los cuales el usuario que lo invoca debe proveer ”inputs” que lo cumplan [7]. Si los ”inputs” permiten la correcta ejecución del contrato inteligente, la transacción es llevada a cabo. En este sentido un contrato inteligente es similar a un contrato legal, pero a diferencia de este último, no es la ley que los hace valer, sino que es la red que los ejecuta de manera automática, transparente e imparcial.

2.2.3 El caso de Ethereum

La idea de programar las transacciones en los contratos inteligentes de Bitcoin produjo alto interés. Debido a esto, nacieron proyectos que extendían funcionalidades de Bitcoin, aprovechando la base de ”opcodes” del sistema. Un ejemplo de esto es ”Mastercoin”, una criptomoneda que utiliza el protocolo de Bitcoin como una capa de abstracción y extendía ciertas funcionalidades, similar a http que ocupa los protocolos de TCP/IP. Este concepto atrae la atención de un programador llamado Vitalik Buterin quien, en el 2013, propone la idea de crear un protocolo que extienda los ”opcodes” de

Bitcoin para poder crear contratos inteligentes de uso general, utilizando la plataforma de Mastercoin [8].

Sin embargo, la propuesta de Vitalik fue rechazada por los desarrolladores de Mastercoin, por lo que este decide crear su propio protocolo llamado ”Ethereum” [13]. Este protocolo describe una máquina virtual que procesa código multi-propósito, la cual es ejecutada por distintos nodos y cuyos resultados son guardados en una cadena de bloques [10].

El protocolo de Ethereum [9], al igual que el de Bitcoin, describe el uso de una red blockchain en donde los nodos consensúan la información de la cadena, mediante el método de ”Proof of Work” y tiene una criptomoneda base llamada ether. Pero, además, Ethereum permite la creación de contratos inteligentes multi-propósito, mediante un lenguaje Turing-completo. Estos contratos son almacenados en la cadena de bloques y pueden ser instanciados por los usuarios, llamándolos a su dirección.

Otra cualidad importante de los contratos inteligentes, es que estos consumen una criptomoneda a medida que se ejecutan, denominada ”gas”. Esto previene que dichos contratos, intencionalmente mal escritos, congelen el funcionamiento de la cadena al ejecutarse (por ejemplo, al contener un ”loop” infinito). Los contratos inteligentes de Ethereum son ejecutados en la ”Ethereum Virtual Machine” (EVM), descrita en la sección 9 de [9], la cual es un entorno completamente aislado del sistema (del nodo que la alberga) y su único trabajo es interpretar el código del contrato inteligente.

2.2.4 EVM y Solidity

La EVM se encarga de interpretar los ”opcodes” del protocolo de Ethereum, si bien estos ”opcodes” permiten codificar cualquier tipo de problema o algoritmo, el entorno de la EVM no permite que se ejecuten infinitamente. Para la ejecución de un contrato inteligente se necesita una cantidad de gas, que depende de la cantidad de instrucciones que el contrato tiene, como se describe en la sección 4 y 5 de [9], la cual también tiene un valor máximo (9820577 al momento de escribir esta memoria [11]). Por lo que la ejecución, no sólo termina cuando llega a un estado ”halt”, sino

que también termina cuando se acaba la cantidad de ”gas” del usuario que llamó al contrato inteligente. Esto implica que existen problemas que, aún siendo debidamente codificados, no podrán ser resueltos por la EVM, al encontrase con un límite teórico finito de tiempo de procesamiento.

Ethereum, además, tiene un lenguaje de programación propio, basado en C++, Python y JavaScript [12] denominado Solidity, el cual es compilado a una representación en bytes del lenguaje ensamblador del protocolo de Ethereum. Solidity es un lenguaje Turing-completo en sí, lo que permite crear contratos inteligentes multipropósitos, aún cuando la EVM puede que no los ejecute debido al límite de gas.

3. ”BLOCKCHAIN” PRIVADOS E HYPERLEDGER

Las mayoría de las cadenas de bloques deben utilizar métodos de consenso que sean tolerantes a fallas bizantinas, porque nodos con comportamiento errático están también permitidos. En el caso de Bitcoin y Ethereum, ambos son protocolos creados y pensados para el uso público [1][13], por lo tanto, su método de consenso se basa en incentivos financieros y competencia entre nodos para asegurar que la mayoría de los nodos se comportarán de forma honesta.

En una red de cadena de bloques es importante, no solo tener un método de consenso que sea tolerante a fallas bizantinas, además debe asegurar una cuota mínima de nodos honestos para que la red cumpla con la condición de tolerancia.

La necesidad de incentivar la honestidad es propio de los sistemas público,s donde los participantes no tienen que identificarse. Sin embargo, los sistemas de cadena de bloques no necesariamente deben ser esquemas públicos.

3.1 Motivación de esquemas privados

El uso de un protocolo basado en una cadena de bloques es motivado por la necesidad de un sistema que asegure confianza, transparencia y descentralización. Por ejemplo, un conglomerado de corporaciones que quieran colaborar entre ellas, compartiendo un activo digital, o corporaciones que quieran certificar su cadena de suministros. En general, entidades públicas o privadas que quieran colaborar entre ellas, pero que no confían completamente las unas a las otras[14] y no quieren someterse a un sistema centralizado donde una asume el control.

Cuando un conglomerado de entidades identificables hace uso de un sistema de

cadena de bloques, puede limitar el acceso de nodos externos y con ellos a nodos maliciosos. Por lo tanto, el uso de métodos de consenso que basen sus incentivos en la competencia entre nodos o activos financieros no son necesarios [14]. Esto es ventajoso porque permite métodos de consensos más flexibles y eficientes que, por ejemplo, métodos como ”Proof of Work” que requieren un alto consumo de recursos computacionales.

3.2 Restricciones y límites de los protocolos privados

Esta libertad de elegir un método de consenso más flexible puede traer consecuencias en la ejecución del sistema. Por lo que en [14] se describe una serie de restricciones a estos sistemas. Entre las más importantes están la ejecución en secuencia y la ejecución no determinista de los contratos inteligentes.

3.2.1 Ejecución en secuencia

Como se describió en el caso de Ethereum, los contratos inteligentes son ejecutados por los nodos hasta que terminan o hasta que se acaba el ”gas” del usuario que instanció el contrato, con tal de prevenir ataques ”DDoS”, en donde un contrato inteligente mal escrito puede intencionalmente congelar el funcionamiento de los nodos y, por lo tanto, de la red. Esto es debido a que los contratos inteligentes se ejecutan secuencialmente [14]. Cada nodo debe ejecutarlos en el orden que son agregados a la red durante el proceso de validación del bloque, es decir, antes que se genere un consenso.

En caso de redes privadas, en donde no existe un activo digital que consumir durante la ejecución de los contratos inteligentes, otros métodos deben utilizarse para evitar el incorrecto funcionamiento del contrato, como por ejemplo, una ventana de tiempo de ejecución máxima fija, una validación de contratos por una cantidad parcial de la red, separar los entornos de ejecución y validación de los nodos, entre otros métodos.

3.2.2 Ejecuciones no deterministas

A veces, el resultado de los contratos inteligentes puede variar según el ambiente de ejecución del nodo. Esto depende del lenguaje en que estos son escritos y la máquina que los interpreta. Por ejemplo, en Ethereum es la EVM la que los ejecuta, la cual es un entorno aislado del sistema [9], y el lenguaje en el que están escritos los contratos está especialmente hecho para evitar el uso de datos volátiles. Por otra parte, si el contrato es escrito en un lenguaje multi-propósito (e.g. Golang, C++, Javascript), ejecutado en el ambiente del nodo, sus resultados pueden diferir si el contrato inteligente incluye comandos que dependen de valores volátiles [14], como resultados de RNGs que ocupan la entropía interna del nodo o posiciones de estructuras de datos no deterministas como los mapas.

Consecuentemente, el diseñador de contratos inteligentes de estos sistemas debe tener presente que, si no es estricto en la codificación, puede tener resultados fallidos.

3.3 Hyperledger

Existen distintos tipos de plataformas que ofrecen soluciones ocupando cadenas de bloques con distintos propósitos y, por lo tanto, distintos métodos y propuestas: desde emular a Ethereum en un ambiente privado, como Quorum [15], hasta sistemas multi-propósito. La mayoría de estos sistemas ocupan un método de consenso basado en PBFT, el cual tiene más probabilidades de éxito al tener cierto control sobre sus nodos, como ocurre en los esquemas privados, y es más rápido en el consenso.

Entre las plataformas privadas existentes está Hyperledger, que es una colección de proyectos relacionados con ”blockchain”. Es decir, Hyperledger es un ecosistema para las cadenas de bloques, como se describe en [16]. Existen proyectos para la creación de sistemas con contratos inteligentes, con cualidades flexibles y basados en la cooperación de organziaciones como Hyperledger Fabric (ver sección 3.4). También está Hyperledger Burrow, que es una emulación de la EVM en un ambiente privado, donde los nodos utilizan Tendermint como método de consenso. Existen herramientas para

monitorear y medir el rendimiento del sistema ”blockchain”, como lo son Hyperledger Explorer e Hyperledger Caliper, entre otros proyectos.

3.4 Hyperledger Fabric

En la sección anterior se habló de las limitantes que pueden tener los sistemas privados de ”blockchain”. Estos inconvenientes son abordados por Hyperledger Fabric (en adelante Fabric) como se le refiere en [18]. Este servicio, al igual que Ethereum, actúa como un computador distribuido que se encarga de ejecutar contratos inteligentes y registrarlos en una cadena de bloques. Sin embargo, tiene una serie de diferencias motivadas por su carácter privado.

3.4.1 Organizaciones, nodos y roles

Como se describe en [18], la red de Fabric es generada por una o varias organizaciones (mediante colaboración). Estas organizaciones heterogéneas aportan a la red distintos nodos. Todos los nodos están identificados por la organización a la cual pertenecen y por su propio identificador, mediante un certificado digital. Este certificado digital puede ser emitido por una entidad externa o un servicio interno de Fabric. Esto significa que no existen nodos anónimos ni nodos externos que puedan participar de la red o acceder sin permiso a los datos escritos en la cadena de bloques. Es la misma red la que decide qué nodos y organizaciones pueden participar. Esto le confiere la característica de red privada a Fabric.

Existen distintos roles entre los nodos, mientras que algunos se encargan de ejecutar los contratos inteligentes, otros pueden acceder a la cadena de bloques y hacer operaciones sobre ella. Otros sirven como nodos intermediarios entre organizaciones y otros participan en el proceso de ordenamiento y generación de bloques. Los nodos pueden tener mas de un rol, simultáneamente, si los roles no son excluyentes.

3.4.2 Chaincode

Los contratos inteligentes en Fabric son llamados ”chaincodes” y hasta le fecha de escritura de esta memoria pueden ser programados en los lenguajes multi-propósito Golang, Java y Javascript (Nodejs), como se señala en la sección 4.5 de [18]. Estos ”chaincodes” son ejecutados por ciertos nodos llamados ”endorsers” o validadores que, a su vez, los ejecutan en una máquina aislada correspondiente a un contenedor Docker [17].

Los chaincodes tienen un formato que debe ser seguido por el programador. Para iniciarlos, tienen una función llama ”Init” que al ser llamada, similar a un constructor de una clase, hace las operaciones iniciales del chaincode, como generar y guardar información necesaria para su funcionamiento normal, iniciar el estado interno, etc. Para hacer uso de las operaciones del chaincode se debe llamar la función ”Invoke”, que es la función que maneja los requerimientos.

Una instancia de Fabric tiene librerías internas que permiten la configuración y el uso de la infraestructura. Estos son[20, 19]:

- LSCC, ”lifestyle system chaincode”: librería estática encargada del proceso en el cual las organizaciones se ponen de acuerdo en cómo operan el chaincode antes de ser usado.
- CSCC, ”Configuration system chaincode”: quien guarda las configuraciones del canal (que se define más abajo).
- QSCC, ”Query system chaincode”: provee la API para las consultas a la cadena de bloques.
- ESCC, ”Endorsement system chaincode”: maneja las políticas de validación de los chaincodes junto al VSCC.
- VSCC, ”Validation system chaincode”: similar al ESCC, operan juntos para manejar las políticas de valdiación de los chaincodes.

3.4.3 Canales, consenso y ordenamiento

Una cualidad de Fabric es que las organizaciones de una red pueden tener distintos ”canales” entre ellos. Un canal es una instancia independiente de una cadena de bloques única (llamada ”ledger”). Así, en una misma red de organizaciones, distintos arreglos de organizaciones pueden tener distintos ”ledgers”, con sus propios ”chaincodes” y reglas que son inalcanzables por el resto de organizaciones de una red. Por ejemplo, en una red R compuesta por las organizaciones $[A, B, C, D] \in R$ pueden existir 3 canales $C1, C2, C3$ con 3 ledgers $L1, L2, L3$ distintos, de la forma:

$$C1(A, C) \leftarrow L1$$

$$C2(A, B, D) \leftarrow L2$$

$$C3(B, C) \leftarrow L3$$

En este ejemplo, B y D no pueden acceder al ”ledger” $L1$ del canal $C1$. Del mismo modo, C no puede acceder al ”ledger” $L2$ del canal $C2$. Por último, A y D no pueden acceder al ”ledger” $L3$ del canal $C3$.

En Fabric, el método de consenso queda a decisión del arquitecto de la infraestructura, ya que, junto a otras características, es un módulo extraíble. Sin embargo, Fabric trae un método por defecto similar a PBFT, como se discute en la sección 2.4 de [18]. Pero, además, las transacciones que corresponden a ejecuciones de un ”chaincode” pasan por distintas etapas. El flujo de las transacciones es explicado en detalle en la sección 3 de [18], el cual consiste en 3 fases:

1. **Fase de Ejecución:** En esta fase, descrita en la sección 3.2 de [18], el cliente propone la transacción a los nodos de validación que simulan la transacción, es decir, ejecutan la transacción en un entorno aislado, pero no guardan los cambios de estado en el ”ledger”. Si la ejecución es correcta, los nodos de validación firman la proposición del cliente, quien recolecta la cantidad necesarias de firmas, según la política del canal. Una vez que esto sucede, el cliente le presenta

la proposición firmada por los validadores al servicio ordenador del sistema.

2. **Fase de Ordenamiento:** Esta fase está descrita en la sección 3.3 de [18]. Una vez que el cliente obtiene las validaciones necesarias, le presenta la transacción al sistema ordenador. Este sistema verifica que todo esté correcto y crea un bloque con las transacciones que tiene en cola. Una vez creado el bloque, el sistema de ordenamiento difunde el bloque entre los participantes de la red para que los nodos la revisen y validen.

3. **Fase de Validación:** Esta última fase está descrita en la sección 3.4 de [18]. Una vez que el bloque es recibido por los nodos, a través del sistema de ordenamiento o por medio de otro nodo, estos lo someten a un último proceso de validación. Primero, verifican que las validaciones cumplen con la política de validación del canal mediante el VSCC. Luego, verifican que no hay un conflicto de escritura, comparando la versión del estado (modificado por la transacción) de cuando fue procesada en la primera fase y el estado actual, evitando así errores de carrera (race-condition). Si todo está en orden, el nodo agrega el bloque a la cadena y actualiza todos los valores locales correspondientes.

3.4.4 MSP y políticas de validación

Existen varios tipos de entidades en Fabric. Además de los nodos y las organizaciones, están los usuarios y los clientes. Un cliente es aquella entidad que puede interactuar con el ”ledger” mediante peticiones al ”chaincode”. En el proceso de 3 fases descritas anteriormente, se habló de que el cliente es aquel que propone una transacción. Para ello, el cliente llama la función de ”invoke”, del ”chaincode”, mediante una petición y agrega en ella todos los argumentos necesarios que la esta requiera. Sin embargo, para que el cliente pueda hacer esta petición, debe tener las credenciales necesarias. En este caso, el cliente hace uso de las credenciales de un usuario que tiene los permisos para operar el ”chaincode”. Dichos permisos pueden ser parciales, es decir, un subconjunto del total de funcionalidades de este. Por otro lado, puede haber un usuario

administrador, que permita el uso de todas las operaciones del ”chaincode”.

Para identificar el tipo de entidad que interactúa en Fabric, se realizan una serie de acciones en un cierto orden. Primero, se verifica si dicha entidad pertenece a las organizaciones de ordenamiento o a las organizaciones de los nodos. Luego, se identifica dicha entidad por la organización específica a la cual pertenece. Habiéndose clasificado según su rol y organización a la cual pertenece, se generan todas las credenciales que necesita. Por ejemplo, si es un nodo verificador, se le asigna una llave privada, un certificado TLS y se le acoplan todos los certificados de las entidades que deba conocer, como los nodos de ordenamiento, el usuario administrador, el certificado de su propia organziación, etc. Todo este esquema de identificación es introducido en [21] como MSP (”Membership Service Provider”), que en realidad no es un ”servicio” como tal (Como lo es OAuth o LDAP), sino que es una pauta de ordenamiento. Cada miembro es identificado y se organiza respecto al esquema del MSP.

Cuando una entidad es identificada mediante el formato del MSP, esta puede hacer las operaciones que le corresponden. La configuración de los permisos es manejada de distintas formas, como son los roles y las políticas de verificación. Estas políticas, descritas en [22], controlan los permisos de escritura, lectura y administración del ”chaincode”. Esto se logra asignando a cada operación una regla que dicta quienes deben firmar el requerimiento para hacerlo válido.

Por ejemplo, en un canal $C1$ en el cual participan 3 organizaciones: A, B, C, donde cada una tiene 2 nodos validadores $[n_{1A}, n_{2A}] \in A, [n_{1B}, n_{2B}] \in B, [n_{1C}, n_{2C}] \in C$, se puede tener una política de escritura sobre el ”ledger” que diga que: ”al menos un miembro de cada organización debe validar la operación”. En ese caso, el cliente buscará que 3 nodos de distintas organizaciones otorguen sus firmas $S(n_X)$ a dicha propuesta P. Basta con $P(S(n_{1C}), S(n_{1A}), S(n_{2B}))$ o cualquier otra combinación de $P(S(n_{iA}), S(n_{iB}), S(n_{iC}))$, con $i \in [1, 2]$, para que la propuesta del cliente cumpla con la política del canal $C1$.

Estas políticas pueden describir reglas tan simples, como la del ejemplo anterior, o reglas mucho más complicadas y específicas. Existen dos formas de crear políticas de

validaciones. En esta memoria sólo se hablará de la forma más común: la ”Siganture-Policy”, descrita en [22]. Esta estructura de datos contiene toda la codificación de la política: qué organizaciones y roles deben tener los firmantes y cuantos de ellos deben firmar. Además, se pueden generar políticas combinadas, como por ejemplo, pedir la firma de ”al menos 1 miembro de cada organización o 2 miembros administradores de cualquier organización”. Esta estructura es cargada (configurada) en la instalación del ”chaincode” o en la configuración del canal.

3.4.5 Infraestructura

Fabric ocupa contenedores Docker para su infraestructura. Esto incluye a los ”chaincodes”. Un nodo validador opera en un contenedor Docker, pero tiene otro (asociado) en donde se ejecuta el ”chaincode” de manera aislada. Por otra parte, un cliente está asociado al menos a un nodo validador, el cual le hace las consultas al ”ledger”. También, cada nodo validador tiene acceso a una base de datos que guarda el estado actual de la cadena de bloques. La cadena de bloques entera es guardada en un sistema de archivos, pero la última realidad (world-state) que describe es guardada en una base de datos para hacer más eficientes la operación sobre ella [23]. Actualmente, existe soporte para dos bases de datos: 1) una simple, denominada LevelDB, que genera un mapa de llaves-valor del estado del ”ledger”, y 2) una base de datos no relacional, más sofisticada, denominada CouchDB, que opera en un contenedor Docker aparte y permite operaciones de búsqueda más complejas.

Un entorno mínimo de Fabric requiere de: una organización con un nodo de validación cuya base de datos es LevelDB, que opera internamente, un cliente, un nodo de ordenamiento y un canal con las políticas por omisión. Un ejemplo de un entorno de 2 organizaciones con 2 nodos cada uno se observa en la figura 3.1.

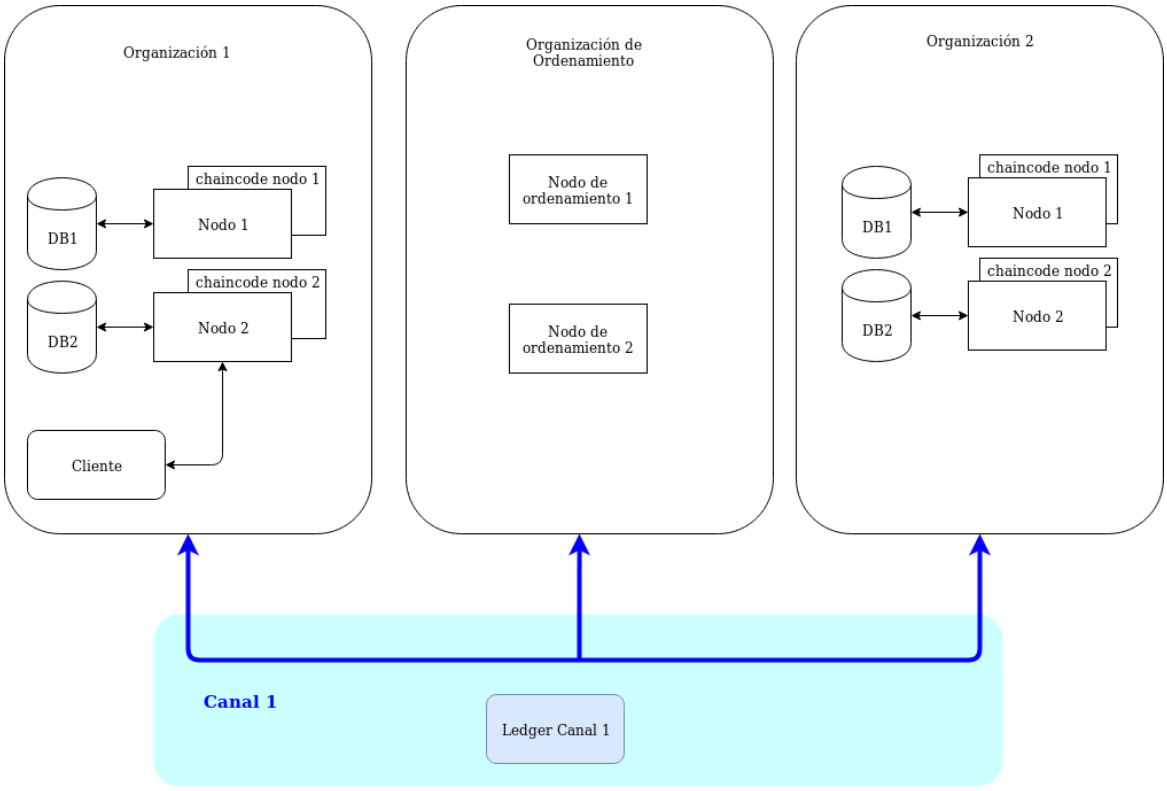


Fig. 3.1: Esquema de Fabric con 2 organizaciones y 2 nodos cada una. El servicio de ordenamiento es una organización aparte, cuyos nodos tienen un trabajo específico. Tanto el ”ledger” como el canal son entidades abstractas que se manifiestan como el consenso de los datos guardados por cada organización.

4. IMPLEMENTACIÓN DE SISTEMA DE I-VOTING EN HYPERLEDGER

Para la infraestructura de la implementación del sistema de votación, se escogió Hyperledger Fabric. En este capítulo se mostrarán los pasos a seguir para configurar, programar y desplegar esta infraestructura.

4.1 Motivación

Un sistema de votación, ante todo, debe presentar un alto sentido democrático en donde los participantes, ya sea por medio de la entidad organizadora o en forma particular, puedan participar, no solo a través del sufragio, sino que ejecutando, fiscalizando y monitoreando el sistema completo.

Esta participación no sólo se extiende a los usuarios, sino que también a las organizaciones formadas dentro de la masa de participantes. Partidos políticos, organizaciones colegiadas, colectivos populares, entidades públicas y privadas de la sociedad civil, entre otras organizaciones podrían participar en las elecciones generales para la presidencia o elecciones municipales. En la misma linea, los centros de estudiantes, organizaciones extra-académicas, departamentos o direcciones académicas podrían participar en las elecciones de Federaciones estudiantiles o de rectoría, como observante y fiscalizadores, aún cuando sus miembros no puedan sufragar. De igual forma, sindicatos, ”holdings”, organizaciones del directorio, entidades fiscales, u organizaciones pertenecientes a los colaboradores, podrían participar en la dirección o decisiones ejecutivas de una corporación.

Mediante Hyperledger, todas estas organizaciones pueden corresponder a una or-

ganización participante en la infraestructura de Fabric. Sólo necesitan una base tecnológica lo suficientemente robusta para formar parte. Si algún particular quiere participar, sin representación, lo puede hacer por medio de una organización neutra. Lo importante es construir las políticas de validación, de tal manera que una organización adversaria no pueda cometer fraude o interrumpir el proceso, al mismo tiempo que la red llega a consenso de manera eficiente. Es por esto que es valorable que organizaciones (ideológicamente) adversarias participen de la red. Estas organizaciones no tienen por qué ser parte del universo de votantes, sólo están como testigos y ejecutores neutros del proceso. Por ejemplo, en una votación para mesa directiva de federación de estudiantes, el cuerpo académico podría participar como observador y garante por medio de una organización con nodos validadores de Fabric, aún cuando los miembros académicos no pueden votar. A su vez, un sindicato puede participar como un nodo ordenador de Fabric, en las elecciones de representantes del directorio de una empresa, aún cuando estos no puedan votar.

Por otra parte, el uso de Fabric permite una composición flexible del contrato inteligente, de tal manera que se adapte a cualquier necesidad particular, propia del proceso de votación. Además, en cuanto al desarrollo con esta plataforma blockchain, existe un amplio soporte y canales de comunicación para obtener ayuda en la creación de proyectos con Fabric.

4.2 Diseño del sistema

Se debe crear un ”software” que adapte los requerimientos de un sistema de votación a la infraestructura de Fabric. La red de Fabric se puede modelar como una nube o una caja negra, con variados estados internos, a la cual se le dan ciertos valores de entrada y se espere ciertas salidas. Con ello se puede diseñar una infraestructura extra, que complemente e incluso reutilice a la de Fabric. La imagen 4.1 ilustra una propuesta de esto.

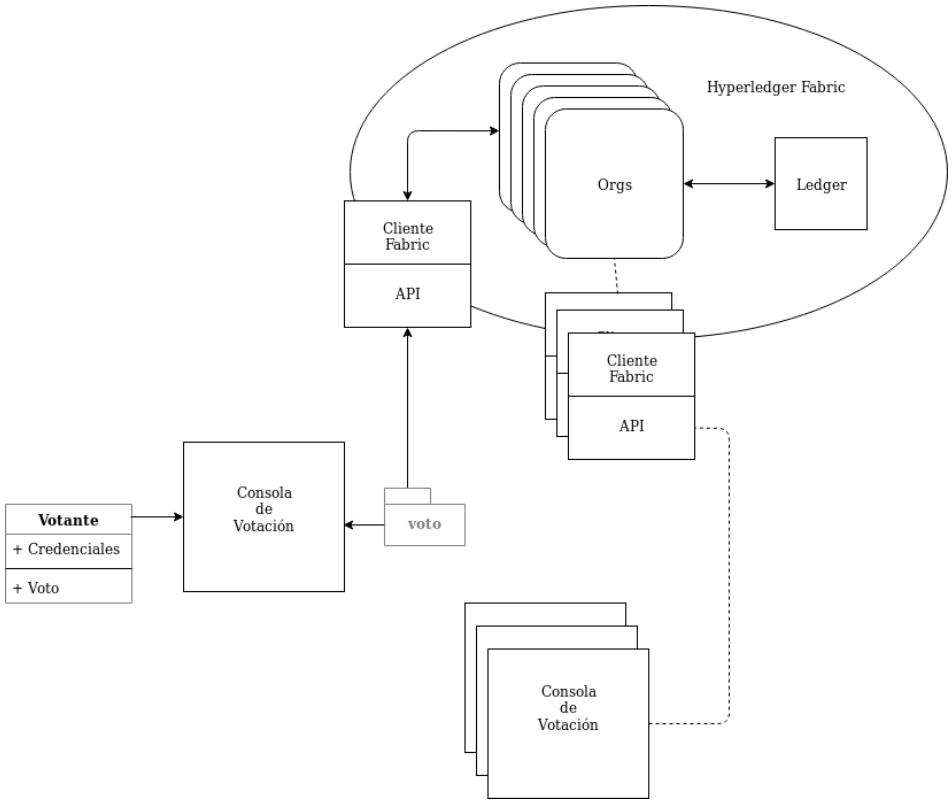


Fig. 4.1: Esquema de la implementación del sistema. Se distinguen 3 softwares: La consola de votación, La API y el ”chaincode” que está presente de forma implícita. Notar que las consolas y los Clientes/API son múltiples pero idénticas.

De la figura 4.1 se desprende que se deben generar 3 ”softwares” los cuales son:

- **Consola de Votación**: La consola de votación se encarga de recibir las credenciales del usuario y su elección y construir un voto válido el cual es entregado a la API de algún cliente de Fabric.
- **API**: La API del cliente que se encarga de recibir los votos de los usuarios y convertirlos en proposiciones a la red de Fabric entre otros requerimientos a la red.
- **”Chaincode”**: El ”chaincode” que procesará las proposiciones y modificará los estados según corresponda.

La consola de votación es un programa sencillo que sólo actúa de interfaz entre el

usuario y el sistema. Esta etapa se centrará en la construcción de la API. Del ”chaincode” se hablará en más adelante.

4.3 Configuración de Hyperledger

Antes de construir la API, se debe construir la infraestructura de Fabric. Para probar el concepto de esta memoria se construyó un piloto. En cuanto a Fabric, este piloto consiste de 3 organizaciones con dos nodos cada uno y un cliente. De estos nodos solo uno es un nodo validador por organización. Se ocupó sólo un nodo ordenador y para la base de datos se ocupó CouchDB.

Se inició mediante el escenario de ejemplo de Hyperledger Fabric ”Build Your First Network” (BYFN). En el proyecto de BYFN vienen las configuraciones y herramientas para la creación de una red con 2 organizaciones con 2 nodos cada uno y un solo nodo ordenador, tal como se detalla en [24]. Por lo que se clonó el repositorio ”Git” y se instalaron todas las herramientas necesarias, como Go, Docker y Docker Composer.

4.3.1 Adaptación de BYFN

Existen 3 archivos de configuración en BYFN:

- **docker-composer-cli.yaml**: El manifiesto para la composición de los contenedores Docker. Prepara la imagen del contenedor, los volúmenes que se cargarán, las variables de entorno del sistema, comando que se ejecuta al arrancar el contenedor, su identificación y su red.

- **configtx.yaml**: El manifiesto que describe la configuración de la infraestructura de Fabric.

- **crypto-config.yaml**: En este manifiesto se configura todo lo referente al MSP. Detalla el formato y el orden de los archivos y directorios referentes a la identificación de las entidades.

Se comenzó por la modificación de los archivos que vienen previamente configurados según la infraestructura de BYFN.

4.3.2 Objetos criptográficos

Para la implementación del piloto existen 2 grupos: El grupo de la organización ordenadora y el grupo de las organizaciones con nodos verificadores. Para el grupo de la organización ordenadora el manifiesto ”crypto-config.yaml” debe definir el nombre, el dominio y las especificaciones del dominio, en este caso su nombre de ”host”. De la forma:

```
OrdererOrgs:
  #---------------------------------
  # Orderer
  # ---------------------------------
  - Name: Orderer
    Domain: example.com

    Specs:
      - Hostname: orderer
```

Para el grupo de las organizaciones verificadoras se debe especificar el nombre de la organización, el dominio, la cantidad de nodos y la cantidad de usuarios. De la forma:

```
# ---------------------------------------------------
# "PeerOrgs" - Definition of organizations managing peer nodes
# ---------------------------------------------------
PeerOrgs:
  - Name: Org1
    Domain: org1.example.com
    EnableNodeOUs: true
    Template:
      Count: 2
    Users:
```

```
    Count: 1

- Name: Org2
  Domain: org2.example.com
  EnableNodeOUs: true
  Template:
    Count: 2
  Users:
    Count: 1

- Name: Org3
  Domain: org3.example.com
  EnableNodeOUs: true
  Template:
    Count: 2
  Users:
    Count: 1
```

La cantidad de nodos se define en el campo *"Template"*. El campo *"EnableNodeOUs"* permite generar políticas de verificación que diferencien las entidades del nodo y del cliente en vez de juntarlas en una sola entidad llamada "OrgMember" como se detalla en la sección "Identity Classification" en [21].

El proyecto "BYFN" trae un "script" binario llamado "cryptogen" que consume este manifiesto YAML y crea los archivos y directorios del MSP [24]. Al ejecutar el comando se generan los siguientes archivos y directorios para el grupo de ordenadores:

```


ordererOrganizations/
 | example.com
     ca
      |    |---- 9a07ffc5a3.....
      |     +---- ca.example.com-cert.pem
      |---- msp
      |     |---- admincerts
```

```
|      |     +---- Admin@example.com-cert.pem
|      |---- cacerts
|      |     +---- ca.example.com-cert.pem
|      +---- tlscacerts
|          +---- tlsca.example.com-cert.pem
```

Que entrega los certificados al dominio y su llave privada para cifrar las comunicaciones.

```

|---- orderers
|     +---- orderer.example.com
|         |---- msp
|         |     |---- admincerts
|         |     |     +---- Admin@example.com-cert.pem
|         |     |---- cacerts
|         |     |     +---- ca.example.com-cert.pem
|         |     |---- keystore
|         |     |     +---- 46185941c753a4.....
|         |     |---- signcerts
|         |     |     +---- orderer.example.com-cert.pem
|         |     +---- tlscacerts
|         |         +---- tlsca.example.com-cert.pem
|         +---- tls
|             |---- ca.crt
|             |---- server.crt
|             +---- server.key
```

Que entrega los certificados al servicio ordenador y su llave privada junto a una llave privada ”TLS” para cifrar las comunicaciones.

```
|---- tlsca
|     |---- 6a89f2c713f2....
|     +---- tlsca.example.com-cert.pem
+---- users
    +---- Admin@example.com
        |---- msp
```

```
|      |---- admincerts
|      |     +---- Admin@example.com-cert.pem
|      |---- cacerts
|      |     +---- ca.example.com-cert.pem
|      |---- keystore
|      |     +---- b0625296daf.....
|      |---- signcerts
|      |     +---- Admin@example.com-cert.pem
|      +---- tlscacerts
|          +---- tlsca.example.com-cert.pem
+---- tls
    |---- ca.crt
    |---- client.crt
    +---- client.key
```

Que entrega los certificados y la llave privada al usuario del ordenador.

La estructura del MSP del grupo de las organizaciones validadoras es similar pero se extiende para las 3 organizaciones, sus nodos y sus usuarios.

4.3.3 configtx.yaml

En este manifiesto YAML se incluye información como: cuántas organizaciones componen la red, sus nombres, sus dominios, sus permisos y la dirección de su información MSP. También se detallan la especificación del sistema de ordenamiento, qué nodos lo componen, sus direcciones, sus permisos, configuración del bloque (tamaño máximo, tiempo de espera para generar el siguiente, cantidad de transacciones, entre otras). Y las configuraciones del canal, quienes lo componen, permisos, etc. Entre otras configuraciones más específicas. Para este piloto las configuraciones más importantes fueron:

```
- &OrdererOrg

    Name: OrdererOrg

```

```
        ID: OrdererMSP

        MSPDir: crypto-config/ordererOrganizations/example.com/msp
        Policies:
            Readers:
                Type: Signature
                Rule: "OR('OrdererMSP.member')"
            Writers:
                Type: Signature
                Rule: "OR('OrdererMSP.member')"
            Admins:
                Type: Signature
                Rule: "OR('OrdererMSP.admin')"
```

Que define al servicio de ordenamiento, notar que el directorio del MSP fue generado con el "script" *cryptogen* mediante el archivo *crypto-config.yaml*. Las políticas de validación son del tipo *"SignaturePolicy"*, introducidas en la sección 3.4.4.

En el caso del resto de las organizaciones la configuración fue similar

```
    - &Org1

        Name: Org1MSP
        ID: Org1MSP
        MSPDir: crypto-config/peerOrganizations/org1.example.com/msp

        Policies:
            Readers:
                Type: Signature
                Rule: "OR('Org1MSP.admin', 'Org1MSP.peer', 'Org1MSP.
    client')"
            Writers:
                Type: Signature
                Rule: "OR('Org1MSP.admin', 'Org1MSP.client')"
            Admins:
                Type: Signature
```

```
                Rule: "OR('Org1MSP.admin')"

        AnchorPeers:
            - Host: peer0.org1.example.com
              Port: 7051
```

En esta configuración se agrega también un nodo ancla. Los nodos de las organizaciones operan en una red privada interna. Estos nodos se comunican con el resto de la red a través del nodo ancla. Esta configuración se replicó para la organización 2 y 3.

Una vez definidas las organizaciones se deben definir las componentes importantes. El sistema de ordenamiento ejecutado por la organización de ordenamiento, definida anteriormente, es de la forma:

```
    Orderer: &OrdererDefaults
    OrdererType: solo
    Addresses:
        - orderer.example.com:7050

    BatchTimeout: 2s
    BatchSize:
        MaxMessageCount: 10

        AbsoluteMaxBytes: 99 MB

        PreferredMaxBytes: 512 KB

    Policies:
        Readers:
            Type: ImplicitMeta
            Rule: "ANY Readers"
        Writers:
            Type: ImplicitMeta
            Rule: "ANY Writers"
        Admins:
```

```
            Type: ImplicitMeta
            Rule: "MAJORITY Admins"
        BlockValidation:
            Type: ImplicitMeta
            Rule: "ANY Writers"
```

Acá se definen el tipo de ordenamiento, su dirección y las características del bloque. Además, para las políticas se usa el tipo *ImplicitMeta*, definidas en [22]. De estas destaca la firma que debe ir en el bloque, la cual pide a cualquier miembro que esté en la categoría de ”Writers”.

4.3.4 Contenedores Docker

Una buena práctica, en el área de infraestructura tecnológica, es mantener los servicios en ambientes independientes de manera de asegurar, planificar y administrar los recursos computacionales, personalizar el ambiente sin interferir en otros servicios, evitar problemas de seguridad, escalar las aplicaciones y mantener orden. Para lograr esto, usualmente, se usan máquinas virtuales que emulan un sistema. Sin embargo, las máquinas virtuales pueden ser innecesariamente pesadas, en términos de costo computacional, y consumir recursos en componentes redundantes, como por ejemplo, el núcleo del sistema. Es de este problema que nace la idea de los contenedores. Un contenedor es una instancia aislada dentro de un sistema que emula a otro pero que reutiliza los recursos del sistema que lo alberga, evitando componentes redundantes.

Uno de los servicios de contenedores más común es Docker. Este servicio reutiliza el núcleo y las librerías de sistema de la máquina que lo alberga, creando un espacio dentro del huésped completamente aislado. Docker ocupa ”imágenes”, que son archivos descriptores de un sistema que, al ejecutarse, crean un ambiente aislado con todas los componentes detallados en la imagen.

La imagen no debe confundirse con el ”Docker File”, que también es un archivo descriptor de contenedores pero de una ejecución en Docker. En general el ”Docker File” contiene las instrucciones de ejecución del sistema: cuál imagen usar, cuál versión,

cuáles variables de entorno, qué usuarios existen, qué volúmenes montar, el comando a utilizar al arrancar, etc.

Fabric ocupa contenedores Docker para los componentes de su infraestructura [25]. Como es una infraestructura amplia, se utiliza una herramienta de Docker llamada Docker Composer, la cual, mediante las imágenes y un archivo de configuración que detalla los componentes, crea toda la infraestructura de manera autónoma, incluyendo la red.

4.3.5 Configuración de Docker Composer

Para configurar la arquitectura del piloto se necesitaron tres archivos. Una base que define un nodo validador y un nodo del sistema de ordenamiento. Un archivo que define todos los nodos de las organizaciones. Y un último archivo de configuración que une todo y define los clientes.

Se inicia con el archivo que define los 2 contenedores bases, uno para los nodos y otro para los nodos ordenadores. La configuración de la base del nodo es la siguiente:

```
1
2 services:
3   peer-base:
4     image: hyperledger/fabric-peer:$IMAGE_TAG
5     environment:
6       - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
7       - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=${COMPOSE_PROJECT_NAME
    }_byfn
8       - FABRIC_LOGGING_SPEC=INFO
9       - CORE_PEER_TLS_ENABLED=true
10      - CORE_PEER_GOSSIP_USELEADERELECTION=true
11      - CORE_LOGGING_GRPC=debug
12      - CORE_PEER_GOSSIP_ORGLEADER=false
13      - CORE_PEER_PROFILE_ENABLED=true
14      - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.
    crt
```

```
15      - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.
    key
16      - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.
    crt
17    working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
18    command: peer node start
```

Existen 4 etiquetas detalladas: la imagen con el nombre ”image”, las variables de entorno del sistema con el nombre ”environment”, el directorio de trabajo con el nombre ”work_dir” y el comando a ejecutar con el nombre ”command”. Se utiliza la imagen que corresponde a la versión de Fabric que se está usando. En este caso, la versión 1.4.2 que puede observarse en [26].

En cuanto a las variables de entorno del sistema, se fijan las direcciones del ”software” de BYFN, las configuraciones de encriptación básicas (No las del MSP) y otras configuraciones menores como el modo de registro de eventos. El directorio de trabajo es en dónde se ejecuta el comando de arranque. El comando de arranque ”peer node start” es la rutina que utiliza Fabric para iniciar la red.

En el caso del servicio de ordenamiento, la configuración es similar, pero se usa la imagen y el comando propias del servicio de ordenamiento y se incluyen variables de entorno como:

```
1       - ORDERER_GENERAL_GENESISMETHOD=file
2       - ORDERER_GENERAL_GENESISFILE=/var/hyperledger/orderer/orderer
    .genesis.block
3       - ORDERER_GENERAL_LOCALMSPID=OrdererMSP
4       - ORDERER_GENERAL_LOCALMSPDIR=/var/hyperledger/orderer/msp
```

Estas variables detallan la configuración para el bloque génesis, el cual es el primer bloque de la cadena de bloques y contiene la configuración inicial de la cadena. Además, se indica la información local del MSP del nodo de ordenamiento.

El segundo archivo de configuración define a todos los nodos de las organizaciones. Parte terminando de definir al nodo de ordenamiento:

```
1 services:
```

```
2
3  orderer.example.com:
4    container_name: orderer.example.com
5    extends:
6      file: peer-base.yaml
7      service: orderer-base
8    volumes:
9        - ../channel-artifacts/genesis.block:/var/hyperledger/
   orderer/orderer.genesis.block
10       - ../crypto-config/ordererOrganizations/example.com/orderers
   /orderer.example.com/msp:/var/hyperledger/orderer/msp
11       - ../crypto-config/ordererOrganizations/example.com/orderers
   /orderer.example.com/tls/:/var/hyperledger/orderer/tls
12       - orderer.example.com:/var/hyperledger/production/orderer
13   ports:
14     - 7050:7050
```

En este archivo se define el nombre de un contenedor individual, como `"orderer.example.com"` y usa el servicio ordenador base como modelo mediante la etiqueta `"extends"` . Luego carga ciertos archivos proveniente de la máquina huésped correspondientes a sus archivos MSP y a su "software". Por último, define los puertos en que opera en la red del huésped.

Luego se define el contendedor de la base de datos como sigue:

```
1  couchdb0:
2    container_name: couchdb0
3    image: hyperledger/fabric-couchdbd
4    environment:
5      - COUCHDB_USER=couch_user
6      - COUCHDB_PASSWORD=couch_password
7    ports:
8      - "5984:5984"
9    networks:
10     - byfn
```

Esta archivo es simple. Define el nombre del contenedor, su imagen, el usuario y contraseña del motor de la base de datos mediante variables de entorno, los puertos expuestos del huésped en el que trabaja y la red a la cual pertenece en Docker. Docker usa una red privada propia para sus contenedores. Al definir que el contenedor estará en la red `"byfn"` hace el contenedor alcanzable para todos los otros contenedores conectados a esta red.

Por último se define al nodo validador:

```
 peer0.org1.example.com:
container_name: peer0.org1.example.com
extends:
  file: peer-base.yaml
  service: peer-base
```

Al igual que el contenedor del nodo de ordenamiento, en este se utiliza la base del nodo de validación como modelo.

```
environment:
  - CORE_PEER_ID=peer0.org1.example.com
  - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
  - CORE_PEER_LISTENADDRESS=0.0.0.0:7051
  - CORE_PEER_CHAINCODEADDRESS=peer0.org1.example.com:7052
  - CORE_PEER_CHAINCODELISTENADDRESS=0.0.0.0:7052
  - CORE_PEER_GOSSIP_BOOTSTRAP=peer0.org1.example.com:7051
  - CORE_PEER_GOSSIP_EXTERNALENDPOINT=localhost:7051
  - CORE_PEER_LOCALMSPID=Org1MSP
  - CORE_VM_DOCKER_ATTACHSTDOUT=true
  - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
  - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdb0:5984
  - CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME=couch_user
  - CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD=couch_password
```

Las variables de entorno describen la configuración del nodo, las direcciones del nodo, en qué dirección local del huésped está, a qué MSP pertenece y su configuración de base de datos.

```
1
2     volumes:
3         - /var/run/:/host/var/run/
4         - ../crypto-config/peerOrganizations/org1.example.com/peers/
    peer0.org1.example.com/msp:/etc/hyperledger/fabric/msp
5         - ../crypto-config/peerOrganizations/org1.example.com/peers/
    peer0.org1.example.com/tls:/etc/hyperledger/fabric/tls
6         - peer0.org1.example.com:/var/hyperledger/production
7     ports:
8       - 7051:7051
9
```

Además, se cargan los archivos importantes como su configuración MSP y el "software" del nodo montando estos volúmenes al sistema. Finalmente se le asigna un puerto del huésped.

Tanto esta configuración, como la de la base de datos, se repica para todos los nodos validadores de todas las organizaciones con sus respectivos nombres y puertos.

Por último en el archivo *"docker-composer-cli.yaml"* se define la red completa:

```
1     networks:
2   byfn:
3
4 services:
5
6   orderer.example.com:
7     extends:
8       file:   base/docker-compose-base.yaml
9       service: orderer.example.com
10    container_name: orderer.example.com
11    networks:
12      - byfn
13
14
15  peer0.org1.example.com:
16    container_name: peer0.org1.example.com
```

```
    extends:
      file:  base/docker-compose-base.yaml
      service: peer0.org1.example.com
    networks:
      - byfn
    depends_on:
      - couchdb0
 .
 .
 .
 .
   peer1.org3.example.com:
    container_name: peer1.org3.example.com
    extends:
      file:  base/docker-compose-base.yaml
      service: peer1.org3.example.com
    networks:
      - byfn
    depends_on:
      - couchdb5

  couchdb0:
    container_name: couchdb0
    extends:
      file:  base/docker-compose-base.yaml
      service: couchdb0
.
.
.
.
 couchdb5:
    container_name: couchdb5
    extends:
      file:  base/docker-compose-base.yaml
```

```
51      service: couchdb
```

Junto a estos contenedores, previamente definidos, se describe también el nodo cliente de la forma:

```
cli:
    container_name: cli
    image: hyperledger/fabric-tools:cli
    tty: true
    stdin_open: true
    environment:
      - GOPATH=/opt/gopath
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      - FABRIC_LOGGING_SPEC=INFO
      - CORE_PEER_ID=cli
      - CORE_PEER_ADDRESS=peer0.org1.example.com:7051
      - CORE_PEER_LOCALMSPID=Org1MSP
      - CORE_PEER_TLS_ENABLED=true
      - CORE_PEER_TLS_CERT_FILE=/opt/gopath/src/github.com/
    hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.
    com/peers/peer0.org1.example.com/tls/server.crt
      - CORE_PEER_TLS_KEY_FILE=/opt/gopath/src/github.com/
    hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.
    com/peers/peer0.org1.example.com/tls/server.key
      - CORE_PEER_TLS_ROOTCERT_FILE=/opt/gopath/src/github.com/
    hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.
    com/peers/peer0.org1.example.com/tls/ca.crt
      - CORE_PEER_MSPCONFIGPATH=/opt/gopath/src/github.com/
    hyperledger/fabric/peer/crypto/peerOrganizations/org1.example.
    com/users/Admin@org1.example.com/msp
```

Además de definir al cliente, se indica que puede tener una sesión ”Bash” mediante una ”tty” y se indican todos los certificados del nodo 1 por el cual el cliente hará las consultas al ”ledger”. Todas estas peticiones serán firmadas mediante los objetos criptográficos del usuario ”Admin” de la organización 1.

```
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
    command: /bin/bash
    volumes:
        - /var/run/:/host/var/run/
        - ./../chaincode/:/opt/gopath/src/github.com/chaincode
        - ./crypto-config:/opt/gopath/src/github.com/hyperledger/
    fabric/peer/crypto/
        - ./scripts:/opt/gopath/src/github.com/hyperledger/fabric/
    peer/scripts/
        - ./channel-artifacts:/opt/gopath/src/github.com/hyperledger
    /fabric/peer/channel-artifacts
        - ../API:/opt/gopath/src/github.com/hyperledger/fabric/API
        - ../API/go-sdk/setup:/opt/gopath/src/go-sdk/
    networks:
      - byfn
    ports:
      - 9999:9999
```

Además de cargar su configuración y la configuración de la red, se cargan los archivos de la API y se indica a qué red pertenece y qué puertos tiene asignados en el huésped.

4.4 Construcción de la aplicación

La red de Fabric maneja los accesos al ”ledger” a través de las políticas y el MSP. Para que un cliente tenga acceso a modificar el estado de la cadena debe tener el usuario con los permisos correspondientes. A su vez los clientes deben tener acceso a la red privada de Fabric para poder hacer las consultas a la cadena.

Por otra parte el cliente debe tener una aplicación pública en la cual los usuarios puedan hacer requerimientos como votar o pedir resultados. Esto fue modelado como una consola de votación y una API. La consola de votación es la interfaz gráfica en la cual los usuarios interactúan con la aplicación y la API contiene la lógica que permite la comunicación con la cadena como se vio en la figura 4.1.

La estructura de la API se presenta en la figura 4.2

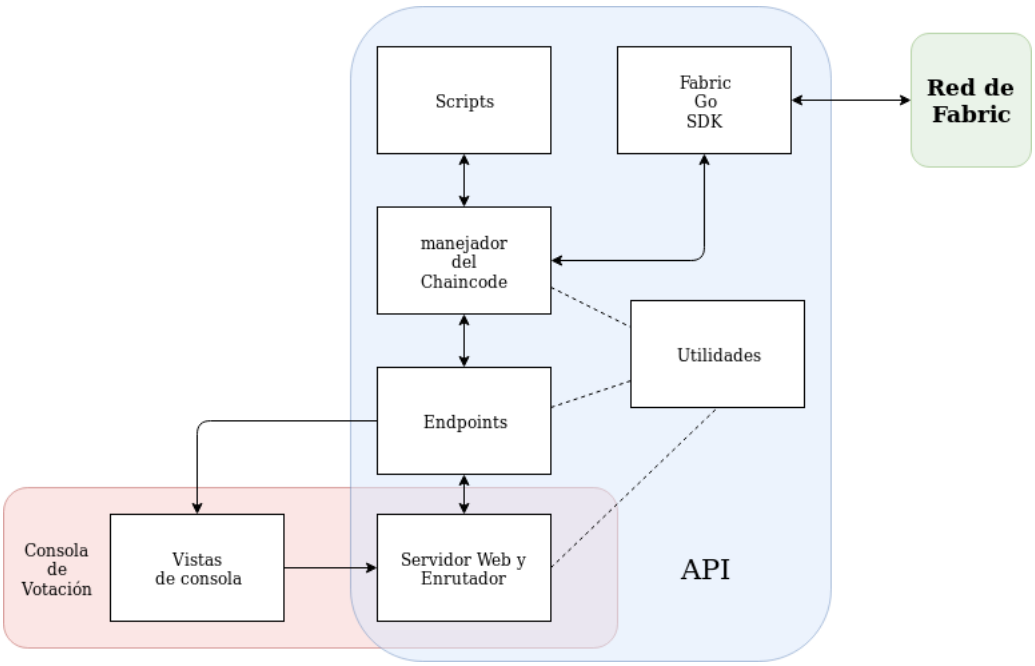


Fig. 4.2: Esquema de la aplicación. Se ven los módulos de la consola de votación y la API. Ambos comparten un módulo servidor y enrutador que recibe las consultas web y las redirige a los módulos que corresponden. El módulo de utilidades puede o no ser utilizado por el resto de los módulos, ofrece funcionalidades simples como formateo, conversión de tipos, respuestas estandarizadas, etc.

4.4.1 Servidor Web

El servidor web descrito en la figura 4.2 es ofrecido por la librería de Go ”Gorilla”, la cual se basa en la librería nativa de Go ”http”. Estas librerías son lo suficientemente robustas para mantener un servidor web sin necesidad de un servicio ”Proxy” con un tráfico moderado. Cada requerimiento es atendido en una ”Go routine”, que corresponde a la funcionalidad de hilos de Golang.

Este servidor web recibe las peticiones a través de las rutas. Las rutas son manejadas por un módulo enrutador. Para el piloto se tienen 6 rutas que se muestran en la figura 4.3. Las primeras dos corresponden al manejo del inicio del ”chaincode”. Las siguientes 2 para el manejo del formulario de votación. Una para obtener los resultados generales y una última para consultar por un voto específico ya realizado.

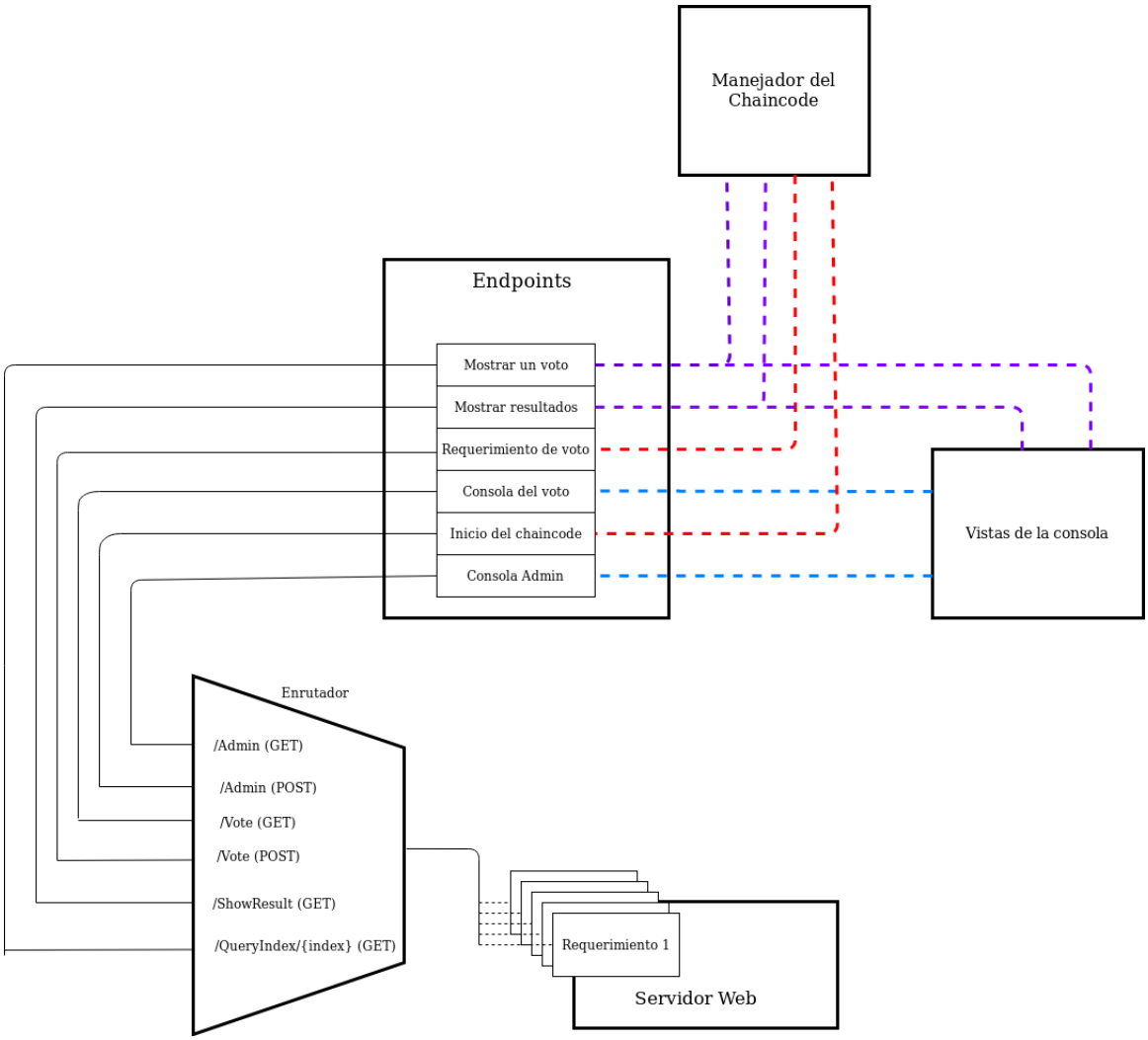


Fig. 4.3: Esquema del servidor Web. El servidor recibe un requerimiento y este es atendido por una sesión aparte. Esta sesión pasa por el enrutador que multiplexa el requerimiento al ”endpoint” que corresponde para que sea atendido.

Como se observa en la figura 4.3, los requerimientos https ”Get” de la consola de administración, que se usa para iniciar el ”chaincode” y el formulario de votación, es atendido por el endpoint que va a buscar el recurso en el módulo de las vistas de las consolas. En el caso de el resultado de votación y la consulta por un voto en específico, el endpoint usa ambos módulos, el de manejo de ”chaincode” y el de la vistas de la consola.

4.4.2 Consola de Votación

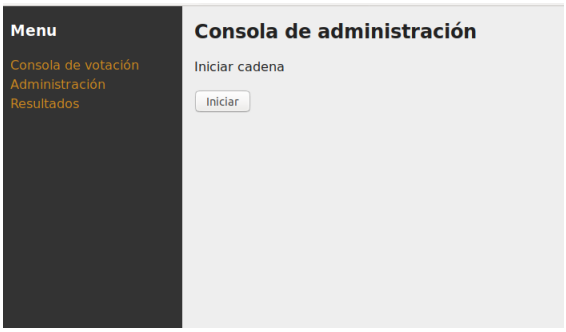
La consola de votación se compone de los módulos compartidos con la APi: el servidor web, los endpoints y el módulo de vistas de la consola. Este último es un módulo que entrega los archivos html de la consola que contiene los componentes para que el usuario interactúe. Estos son 3 y se pueden apreciar en la figura 4.4:

- **Admin**: Esta vista es llamada a través de la ruta ”/Admin (GET)” y su contenido puede apreciarse en la figura 4.4a donde se ve un botón que al ser presionado llama a la ruta ”/Admin” (POST) de la API, donde ”GET” y ”POST” son los tipo de requerimiento http.

- **Vote**: Esta vista es llamada a través de la ruta ”/Vote (GET)” y contiene 2 formularios, como se ve en la figura 4.4b. En el primero, el votante puede realizar su voto agregando su credencial de votante y seleccionando su candidato de preferencia. Al oprimir ”enviar”, una función escrita en javascript dentro del código html envía un requerimiento con los valores ingresados a la ruta ”/Vote (POST)” de la API.

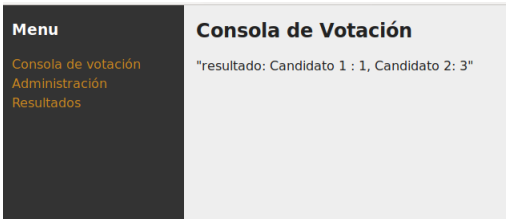
  El otro formulario permite conocer la información de un voto. El votante inserta el número identificador del voto. Luego de presionar ”consultar”, se envía un formulario a la ruta ”/QueryIndex/index” de la API, donde ”index” es el número del voto consultado. Los datos del voto son mostrados en la parte inferior de esta vista.

- **Results**: Esta vista es llamada mediante la ruta ”/ShowResults” de la API y muestra los resultados parciales de la votación, como se aprecia en la figura 4.4c.



(a) Inicio del ”chaincode”

(b) formulario de votación



(c) Resultados

Fig. 4.4: Vistas de la consola de votación. En el formulario de ”Consultar por Voto”, ”nullifier” se refiere al índice del voto. Este concepto será estudiado más adelante.

4.4.3 API

Como se aprecia en la figura 4.2, la API consta de 6 módulos. Ya se vio que el módulo del servidor web recibe los requerimiento y el enrutador los multiplexa al ”endpoint” correspondiente, como se ve en la figura 4.3. Pero, además, la API cuenta con otros 4 módulos que son:

- El manejador del ”chaincode”.
- Los ”scripts”.
- El ”Software Develompment Kit” (SDK) de Fabric para Golang.
- Las utilidades.

Manejador del Chaincode

El manejador del ”chaincode” contiene 5 funciones que hacen requerimientos al ”chaincode”. La primera llamada ”Setup” se encarga de generar todo los datos necesarios para dar inicio a la votación, como lo son los candidatos, el padrón electoral y otros meta-datos. Una vez generada esta información, se incorpora en un requerimiento y es enviado a la red para inicializar con estos datos el ”chaincode” entre los nodos.

Las siguientes dos funciones llamadas ”MakeProof” y ”Vote” son las encargadas de hacer el requerimiento del voto. La primera busca la información del proceso de votación necesario para generar, junto a las credenciales del usuario, una prueba que valide al votante. Este proceso será detallado mejor en los próximos capítulos. Una vez generada la prueba, se llama a la segunda función que junta esta prueba, su identificador y el valor del voto para crear un requerimiento de voto a la red.

Las otras dos funciones corresponde a las ”queries” que se la hace a la base de datos de la cadena para conocer los resultados parciales de votación y consultar un voto en específico.

Scripts

Los ”scripts” son programas de sistema que ejecutan ciertas herramientas, principalmente criptográficas, que ayudan en la construcción de los votos y del padrón electoral.

Los ”script” escritos en Bash llaman una herramienta especial que construye las pruebas de los votos y un script escrito en Python llama a una librería que permite

generar llaves y firmas digitales en un esquema de encriptación específico.

Estos ”scripts” son llamados en los endpoints mediante la librería ”exec” nativa de Golang que se encarga de leer la salida estándar de su ejecución.

Fabric Golang SDK

Para la interacción con Fabric se dispone de un SDK que es una colección de librerías escritas en distintos lenguajes. Existen librerías para Java, NodeJS, Golang y Python.

Para este proyecto el SDK elegido fue el escrito en Golang y se utilizó para las siguientes tareas:

- Inicializar el ”chaincode”
- Hacer requerimientos de lectura a la base de datos.
- Hacer requerimientos al ”chaincode” y modificar su estado

Para ello se necesitó librerías que ayudaran a definir un canal y un cliente. Definir las políticas del ”chaincode”. Construir objetos de conexión a los nodos validadores y utilidades que ayudaran a formatear los requerimientos al estándar requerido por Fabric.

El módulo de SDK es utilizado por el módulo de manejo del ”chaincode” para hacer los requerimientos a la red con los procedimientos requerido por los protocolos de Fabric. Para ello, se crea un objeto que es capaz de cargar un archivo de configuración con toda la información relevante respecto a la arquitectura del piloto de Fabric. Este objeto tiene los métodos necesarios para hacer acciones sobre el ”ledger”.

Por lo tanto se define un objeto de la forma:

```
type FabMaster struct {
  ConFile    string
  ChainCode  string
  Channel    string
  IsInit     bool
```

```
  sdk         *fabsdk.FabricSDK
  OrgID       string
  User        string
}  ,
```

donde

- ”Confile” es el nombre del archivo que describe toda la red y las configuraciones del piloto.
- ”Chaincode” es el nombre del ”chaincode”.
- ”Channel” el nombre del canal.
- ”OrgID” , el nombre de la organización a la cual pertenece el cliente.
- ”User” el nombre del usuario que utiliza el cliente para las consultas.
- ”IsInit” es un ”booleano” que indica si la red ya ha sido iniciada o no.
- ”sdk” es el objeto proveniente de la librería del SDK que permite acceder a las herramientas de conexión.

La estructura tiene métodos que permiten obtener la información del canal y la del cliente. Permite tener los objetos criptográficos del MSP del cliente y los certificados de las entidades del canal. Obtener el nombre de los nodos validadores y las políticas del ”chaincode”.

Con la ayuda de esta estructura se pueden crear funciones para hacer acciones en la red como se detalló al principio de la sección 4.4.3 las cuales serán detalladas más adelante cuando se defina el ”chaincode”.

Utilidades

Por último el paquete de utilidades ofrece rutinas básicas repetitivas que son útiles en gran parte de los módulos de la API, como la definición de constantes, la serialización de estructuras JSON, el formateo de respuestas http, etc.

4.5 Puesta en marcha de Hyperledger

Definidas todas las configuraciones de la sección 4.3.3 y programada la aplicación de la sección 4.4, lo último que queda por hacer es preparar el ambiente para que la aplicación funcione en el cliente y levantar toda la red.

La aplicación requiere de librerías que no vienen por defecto en Golang, además de las librerías y herramientas (las cuales se discutirán más adelante) usadas por lo scripts. Estos paquetes son instalados en el contenedor del cliente, para evitar que el contenedor requiera acceso a internet para poder crearse. Una vez creado el ambiente necesario para que la aplicación funcione, el estado actual del contenedor es convertido en su imagen. Cada contenedor Docker ocupa una imagen en la cual se basa, como se explicó en la sección 4.3.4. Sin embargo, al contenedor se le pueden hacer cambios como: cambiar el contenido de archivos y directorios, instalar paquetes, cambiar sus variables de entorno, etc. Cuando el contenedor es eliminado todos estos cambios desaparecen y el nuevo contenedor trae las configuraciones de su imagen y de su archivo descriptor (llamado ”Docker File”, en el caso de no ocupar Docker Composer). Para mantener estos cambios en la imagen, el estado actual del contenedor puede ser guardado (en la imagen) mediante el comando ”Docker commit”.

Una vez guardado el estado necesario para que la aplicación funcione en la imagen del cliente, se procede a poner en marcha la red.

4.5.1 Script de arranque

BYFN tiene un ”script” escrito en Bash, que realiza toda la rutina necesaria para que Fabric funcione. Esta rutina ejecuta las siguientes tareas:

- Genera todos los elementos criptográficos incluyendo los certificados TLS y el material del MSP.
- Genera los contenedores Docker declarados en el archivo de configuración del Docker Composer.

- Crea el bloque génesis y toda la configuración del canal.
- Instala el ”chaincode” en los nodos validadores. Crear y agregar los nodos al canal.

Para la primera tarea se apoya en el comando:

```
cryptogen generate --config=./crypto-config.yaml
```

Donde el archivo ”crypto-config.yaml” contiene la configuración descrita en 4.3.2. Al ejecutarse este ”script” todos el árbol de archivos descrito en la sección 4.3.2 es generado.

La segunda tarea es realizada mediante el comando:

```
docker-compose -f docker-composer-cli.yaml -f up -d
```

Donde ”docker-composer-cli.yaml” es el tercer archivo descrito en la sección 4.3.5. Al ejecutar este comando todos los contenedores declarados en el archivo son creados.

La tercera tarea es realizada con los comandos:

```
configtxgen -profile PilotChannel -channelID byfn-sys-channel -
    outputBlock ./channel-artifacts/genesis.block

configtxgen -profile PilotChannel -outputCreateChannelTx ./channel-
    artifacts/channel.tx -channelID $CHANNEL_NAME
```

El primer comando corresponde a la generación del bloque génesis de la cadena de bloques y, el segundo, a la configuración del canal. El perfil ”PilotChannel” corresponde al perfil de configuraciones descrito en la sección 4.3.3 (3 organizaciones con 2 nodos cada una y un solo nodo de ordenamiento).

La última tarea es realizada mediante los comandos:

```
peer channel create -o orderer.example.com:7050 -c $CHANNEL_NAME -f
    ./channel-artifacts/channel.tx

peer channel join -b $CHANNEL_NAME.block

```

```
5 peer chaincode install -n evmcc -v ${VERSION} -l go -p ${CC_SRC_PATH
    }
```

Todos estos comandos son ejecutados en el contenedor de todos los nodos validadores. los archivos *"./channel-artifacts/channel.tx"* y *$CHANNEL_NAME.block* fueron generados en las tareas anteriores. Una vez que el "script" es terminado se aprecia un mensaje como el de la figura 4.5. Los contenedores pueden ser observados funcionando en la figura 4.6. Con esto la red Hyperledger Fabric queda operativa.

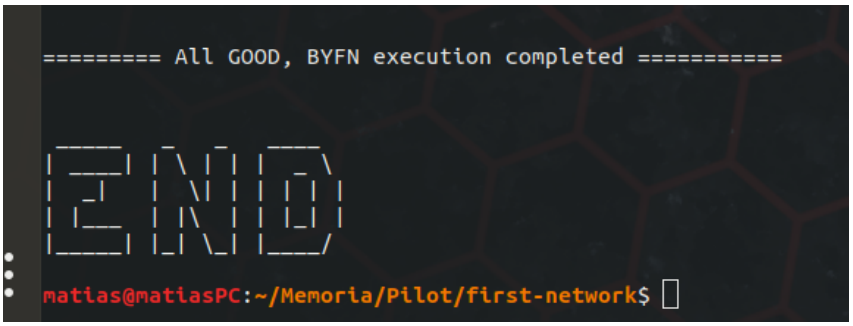


Fig. 4.5: Mensaje de éxito del "script" de BYFN

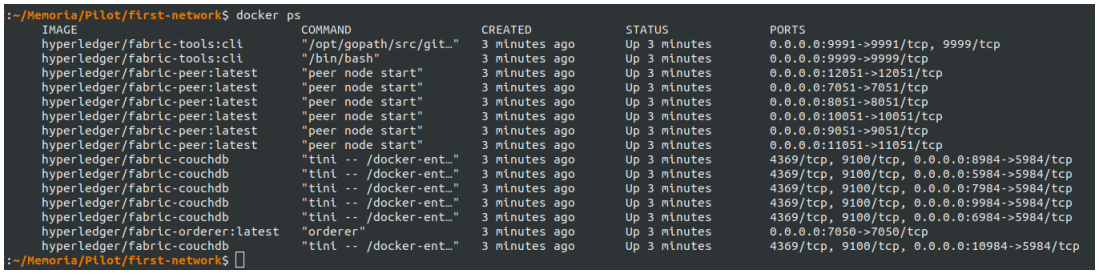


Fig. 4.6: Contenedores Docker después del "script" de BYFN. En la imagen se aprecian 6 contenedores de nodos validadores (2 por cada una de las 3 organizaciones), 6 contenedores de CouchDB, una para cada nodo. Un nodo ordenador.

5. ANONIMATO Y PRUEBAS DE CERO CONOCIMIENTO

En un sistema convencional de voto el anonimato es conseguido al asegurarle al votante que nadie es testigo de su elección y la papeleta se pierde entre otras similares. Solo se verifica la identidad del votante al momento de presentarse ante un vocal, de manera de comprobar que está habilitado para sufragar. Luego de esto, su identidad no importa y su voto se vuelve indistinguible del resto. Por lo que no hay forma de vincular un voto con un votante. Esto funciona ya que se asume que *todo* voto en la urna es válido.

En un sistema digital esto es difícil de conseguir, no sólo porque los datos en tránsito en un canal se pueden replicar y modificar, sino que es difícil verificar la habilidad de sufragar de una persona sin usar una identidad digital que perdure. Sin acoplar la identidad con los votos no se puede tener una confianza estricta en ellos como en el mecanismo convencional.

Una vez que una persona sube a una red su identidad, esta pierde el control de lo que se haga con ella. Puede que sus credenciales sean interceptadas por terceros, o guardadas en un lugar que no es de confianza. Incluso, si se le asegura a una persona que su identidad no será revelada intencionalmente, la entidad a cargo de cuidarlas puede ser atacada o cohechada de manera que las filtren. En general, el hecho de que se deba vincular la identidad de un votante con su voto genera un riesgo al anonimato, incluso si su custodia es protegida.

Es por lo anterior que se debe encontrar un mecanismo en donde se pruebe que un voto proviene de una persona habilitada para votar, sin que esta pierda el control de su identidad.

5.1 Identidad digital

Un sistema de votación tiene un padrón electoral. El sistema debe ser capaz de identificar a los votantes habilitados mediante este padrón y un identificador proporcionado por los votantes. Este identificador debe poder ser usado solo por ellos. Además debe ser digitalizado de manera que pueda operar en el sistema. Es por ello que, en este sistema de votación, los votantes tendrán una identidad digital que les permita emitir votos válidos.

Para estos efectos a cada votante se le asignará un par de valores. Uno público -que lo identifique- y uno privado -que le permita hacer uso exclusivo de esta identidad.

5.1.1 Secreto, preimagen e Imagen de una función Hash

En comunicaciones digitales se prueba la identidad de una persona presentando sus credenciales. El esquema general es la presentación de una contraseña al sistema en un canal protegido. Si la contraseña se corresponde a un registro relacionado, e.g. la transformada $T(contrasena)$ guardada en el sistema de autenticación, entonces se verifica que la persona está registrada en el sistema. Luego se le otorgan los privilegios de la identidad correspondientes a $T(contrasena)$.

Para que este esquema sea seguro se necesita que la transformada $T()$ proteja la contraseña y la contraseña debe, en todo momento, permanecer en secreto. Para ello se puede usar un función "Hash" $H()$ como $T()$, ya que no se puede deducir la preimagen de una buena función "Hash" conociendo su imagen.

Con ello la imagen "S" de la función "Hash" corresponde a la identidad de una persona, y el secreto "s" corresponde a la pre-imagen la cual le da los privilegios de usar dicha identidad una vez presentados y verificado que $H(s) = S$. Cabe destacar que, si s y $H()$ son lo suficientemente robustos, se puede publicar S sin comprometer la seguridad del sistema. Lo anterior se puede apreciar en la figura 5.1

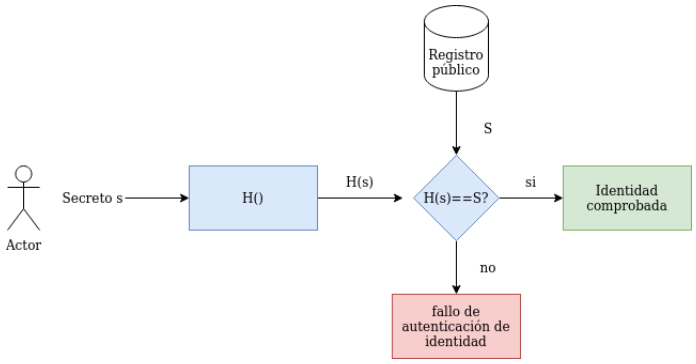


Fig. 5.1: Esquema del sistema de identificación. El secreto s sólo es conocido por el usuario mientras que su identificador S, producto del "Hash" de s, puede ser guardado en un repositorio público y usado en un sistema de autenticación básico.

5.2 Pruebas de Cero Conocimiento

En [27] Goldwasser, Micali y Rackoff introdujeron una técnica para probar de forma interactiva, probabilística y eficiente algún teorema. Pero también plantea la siguiente interrogante: ¿Cuánto conocimiento se debe comunicar de una prueba del teorema?. Con esta interrogante plantean un sistema que oculta cierta información del teorema, pero que de todos modos lo prueba. Este concepto ha sido ampliamente discutido y se le da el nombre de pruebas de cero conocimiento (ZKP por sus siglas en inglés : *"Zero Knwoledge Proofs"*).

En un esquema de comunicación, donde una entidad A quiere demostrar la existencia de un valor z a otra entidad B, sin revelar su valor mediante una ZKP, el protocolo de la ZKP cumplirá con las siguientes 3 propiedades -como se detalla en [28]:

- **Completitud**: Si A y B son honestos, A le podrá demostrar la existencia de z a B.
- **Robustez**: Si no existe z, A no podrá convencer a B de la existencia de este.
- **Cero Conocimiento**: Al final del proceso, B no sabrá el valor de z ni nada que lo ayude a deducirlo.

En general las ZKP suelen recurrir al esquema:

$$C(\vec{x}, \vec{z}) = \vec{Y} \tag{5.1}$$

Donde $(\vec{x})$ es el valor o los valores de entrada, públicos, y $\vec{z}$ son los valores cuyo conocimiento se quiere demostrar sin revelarlos, llamado testigo o testigos (witness). $C()$ es una computación arbitraria que consume los valores $\vec{x}$ y $\vec{z}$. $\vec{Y}$ son valores de salida, públicos, que cumplen con la ecuación 5.1.

Un protocolo ZKP, entonces, tiene al menos dos pasos:
Generar una prueba:

$$P = G(C, \vec{x}, \vec{z}, \vec{Y})$$

Verificar una prueba:

$$V(P, C, \vec{x})$$

En la cual V es una función de verificación que retorna un estado de éxito o fracaso. Por ejemplo, un valor booleano ”verdadero” en caso de ser una prueba verídica. Nótese que, en el paso de verificación no se utilizan los valores de $\vec{z}$, y que ni P, C o $\vec{x}$ revelan algo de $\vec{z}$.

La formulación matemática detrás de los esquemas de los ZKP suele ser compleja, además de requerir que distintas técnicas trabajen en conjunto [28]. Estas técnicas varían según el protocolo que se use y, por lo tanto, varían sus propiedades y el tipo de computaciones y testigos que se pueden usar. Existen ZKP que son especialmente buenas para demostrar la pertenencia de un valor z a un rango de números. Otros métodos que son bastante eficientes, pero requieren la participación tanto de quien quiere verificar como quien produjo la prueba en el proceso de validación, conocidas como pruebas interactivas [27] y otras más modernas que no requieren interacción con el autor de la prueba [29]. Existen otras pruebas que son probabilísticas [27] y otras que son deterministas, es decir, que al aplicarse el proceso de verificación, el receptor

queda completamente convencido. Todo estos tipos de características dependen del protocolo, por lo que se debe elegir uno que se adecue a la naturaleza del sistema al cual se quiere aplicar.

Teniendo en cuenta esto, se puede usar un protocolo de ZKP que pruebe que un votante posee un secreto s que, junto a una función Hash H verifica:

```
C = V(s,S){
     H(s) == S
}

Z = s
X = S
Y = H(s)==S?:1,0
P = G(C,X,Z,Y)
```

Donde lel testigo es s, S es el valor público y la computación C es una función V que verifica el conocimiento de la pre-imagen. La prueba P es generada por un proceso G usando a los testigos, la computación y el valor público.

Este esquema verifica que el votante posee un secreto, pero no demuestra que dicho secreto esté inscrito en el padrón electoral. Para poder demostrar la inscripción, una computación C más extensa debe utilizarse. Sin embargo, se deben explorar los alcances de los sistemas ZKP a disposición para construir una computación C que permita verificar la validez del votante.

La definición que se tiene hasta el momento de las ZKP es bastante amplia. Pero no todos los esquemas conocidos son aplicables a un sistema de votación. Se debe considerar que este tipo de sistemas requerirá una generación, transporte en red y verificación de múltiples ZKPs. Por lo tanto, el protocolo elegido debe ser capaz de lograr crear y verificar ZKPs en tiempos razonables y con tamaños prudentes, para que puedan transportarse. Además, requerirán que sean autosuficientes, es decir, que no requieran una interacción con el autor de la prueba para su verificación.

5.3 ZKPs no interactivas

5.3.1 Fuentes de aleatoriedad secretas

Los primeros esquemas de ZKP requerían la interacción del emisor y del verificador. En [27] y [29] se describe cómo estos sistemas funcionan. Un verificador B emite números aleatorios y el emisor A debía responder según el valor de estos números, algo que demostrara el conocimiento de una proposición L, sin revelar nada que lo descubriera.

Por ejemplo. A y B son personas que están en una misma habitación y A quiere convencer a B que tiene un poder mágico oculto que no quiere revelar y que le permite saber cuando una persona cambia un objeto de mano. Por lo que B realiza el siguiente experimento: Toma una moneda con una mano. Se pone en frente de A. Pone sus dos manos en su espalda y aleatoriamente elige si cambia la moneda de mano o no. Luego le muestra los dos puños cerrados a A y le pregunta "¿En qué mano está la moneda?".

Si A adivina la primera vez, puede que en realidad haya tenido suerte al enfrentarse en una situación donde tiene 50 % de probabilidad de acertar. Sin embargo, tiene $(0.5)^k * 100$ % de acertar k veces seguidas. Por ejemplo A tiene 0.0976 % de probabilidades de acertar 10 veces seguidas, a menos *que realmente tenga el poder mágico secreto que dice tener*. En este esquema de prueba, B repetirá el experimento las veces que sea necesaria hasta que se convenza.

Con este ejemplo se puede ver que B requiere *interactuar* con A para que el esquema funcione. Esta interacción se basa en que B decide aleatoriamente si cambiar o no de mano la moneda, por lo que A reacciona a un resultado aleatorio que *no puede anticipar*. Si, por ejemplo, en la misma habitación en donde se encontraban A y B estuviera una persona C a la cual B le hubiera contado con anticipación cuántos experimentos haría y en cuales cambiaría la moneda de mano y a su vez este le informara de esto, de alguna manera a A, sin que B se diera cuenta, A podría convencer a B de que tiene un poder mágico sin realmente tenerlo.

5.3.2 Esquemas no interactivos

En las pruebas interactivas los verificadores usan una fuente de aleatoriedad que no puede ser conocida por los emisores. Si se utiliza un esquema distinto en donde la fuente de aleatoriedad secreta no es necesaria y en cambio se utiliza un oráculo o un emisor de aleatoriedad público, entonces, la interacción con el emisor A tampoco es necesaria. En el mismo ejemplo anterior, si A escribiera en una nota los resultados de los experimentos *antes* que B los realizara y luego pusiera esta nota en un sobre cerrado, B podría verificar que A posee el poder mágico al realizar los experimentos en solitario, sin abrir el sobre y luego comparar, al final, los resultados con los escritos en la nota. En este caso, da lo mismo si A tiene acceso a la fuente de aleatoriedad durante el experimento, ya que la nota con sus resultados ya estaba escrita y por lo tanto tampoco se requiere la interacción con él.

Como se menciona en [29], los esquemas no interactivos son ventajosos porque reducen el número de mensajes en la red. En el ejemplo anterior, donde B realiza 10 experimentos, significa que se necesitan 10 mensajes de consulta y 10 mensajes de respuestas con el emisor para cada prueba y para cada verificador. Es decir una red que utiliza ZKP interactivas con 10 experimentos de verificación por ronda, requiere 20 veces más mensajes que una que utiliza pruebas no interactivas. Esto es significativo para redes con una gran cantidad de clientes y nodos verificadores.

6. ZK-SNARK

En el capítulo anterior se introdujeron las ZKPs. Entre estas se mencionó lo ventajoso que son los esquemas no interactivos y también se mencionó la necesidad de encontrar un esquema que genere y valide pruebas en un tiempo prudente y cuyo tamaño permita su transporte en la red.

Entre las distintos esquemas que existen están las ZK-SNARKs, cuyo acrónimo significa *"Zero Knowledge - Succint Non-interactive ARguments of Knowledge"* o, en español, "Argumento de Conocimiento Sucinto No interactivo de cero conocimiento". Este nombre es bastante específico y descriptivo en la nomenclatura de las ZKPs y significa:

- **Argumento de Conocimiento**: Además de probar la existencia de cierto valor x, prueba que el emisor conoce o posee dicho valor.
- **Sucinto**: El tamaño de la prueba es fijo y puede procesarse en tiempo polinomial.
- **No interactivo**: No requiere ronda de mensajes con el emisor de la prueba para poder verificarse.
- **Cero Conocimiento**: El verificador no aprende nada más, aparte de que el emisor conoce a x.

6.1 Fundamentos matemáticos de los ZK-SNARK

Para que una ZK-SNARK funcione, se requiere que distintas técnicas matemáticas operen en conjunto. Es importante que se entiendan alguna de estas para poder com-

prender la mecánica de funcionamiento, las ventajas y restricciones de este protocolo.

6.1.1 Programa Aritmético Cuadrático

Como se ve en la ecuación 5.1, se necesita de una computación $C()$ para generar una prueba. Las ZK-SNARKs operan sobre un tipo de computación especial. El programa $C()$ debe formularse como un Programa Aritmético Cuadrático (QAP, por sus siglas en inglés).

Un QAP es una representación polinómica verificable de un programa escrito en algún lenguaje L, Turing completo, o lo que es equivalente, a un lenguaje perteneciente a NP, como se define en la sección 1.1 de [32]. La definición de una QAP se puede apreciar en la sección 7.1 de [32], que dice:

Un QAP Q sobre un campo $\mathbb{F}$ contiene 3 grupos de polinomios $V = \{v_k(x)\,, k \in \{0, ..., m\}\}$, $W = \{w_k(x)\,, k \in \{0, ..., m\}\}$, $Y = \{y_k(x)\,, k \in \{0, ..., m\}\}$ y un polinomio $t(x)$ que, para un conjunto de valores $\vec{a} = (a_1, a_2, a_3...a_m)$ correspondiente a las entradas con índices $1..., n$ y salidas con índices $n' + 1, ..., m$ de la computación $C()$, cumple con:

$$t(x) \quad divide \quad \Big(v_0(x)+\sum_{k=1}^{m} a_k*v_k(x)\Big)*\Big(w_0(x)+\sum_{k=1}^{m} a_k*w_k(x)\Big)-\Big(y_0(x)+\sum_{k=1}^{m} a_k*y_k(x)\Big) \tag{6.1}$$

o, equivalentemente:

$$P(x) =< \vec{a}, V(x) > * < \vec{a}, W(x) > - < \vec{a}, Y(x) >= t(x)H(x)\,, \tag{6.2}$$

donde $< a, b >$ denota el producto interno de "a" y "b".

Hasta el momento, un emisor A que conoce unos valores $\vec{a}$ que satisfacen $C(\vec{x}, \vec{z}) = \vec{Y}\,; \quad (\vec{x}, \vec{z}, \vec{Y}) \in \vec{a}$ debe convertir $C()$ en a la forma de la ecuación 6.1. Luego, junto a los grupos de polinomios V, W, Y y al polinomio $t(x)$ compondrá la ecuación 6.2,

si los valores $(a_1, a_2, a_3, ..., a_n, a_n + 1...a_m)$ resuelven la computación $C()$. Entonces, $t(x)$ dividirá a $P(x)$ sin dejar resto.

Para construir un QAP a partir de una computación $C()$ se debe pasar por un proceso consistente en varios pasos.

6.1.2 Proceso de creación de un QAP

Primero, se debe convertir la computación $C()$ en un circuito aritmético. Un circuito aritmético corresponde a una representación mediante compuertas aritméticas, conectadas entre si, de un problema matemático más complejo. Una compuerta aritmética tiene dos entradas, una operación interna y una salida. Las dos entradas son los dos operando y la salida es el resultado. Los operandos permitidos son (+,-,*,/), así se pueden construir operaciones como x^2 mediante la multiplicación $x * x$, pero no se admiten operaciones como 2^x en una compuerta. El proceso de convertir un programa en un circuito aritmético se conoce como ”flattening” (Aplastamiento) que se basa en operaciones de un solo operando apoyado en variables auxiliares. Por ejemplo:

Sea $C(x)$ Una computación que demuestra la existencia de un valor x que cumple con $x^2 + 6x + 13 == 4$. El proceso de ”flattening” es:

Primero se define una función *eval*:

```
def eval(x){
    y= x*x
    return y+6x+13
}

```

Luego se inicia el proceso de ”flattening” sobre *eval(x)*

```
y = x*x
aux_1 = 6*x
aux_2 = y+aux_1
~out = aux_2+13

```

Del proceso anterior se obtienen 4 compuertas aritméticas, de las cuales la variable de salida " out" debe verificar que sea igual a "4" para una entrada x.

Luego de encontrar el circuito aritmético, se procede al siguiente paso. Traspasar el circuito encontrado al formato R1CS [37]. El formato de "Sistema de Restricción de Rango 1" (R1CS, por sus siglas en inglés "*Rank 1 Constrains System*") es un formato que reconstruye el circuito aritmético mediante productos internos de 3 vectores, de la forma detallada a continuación.

Sean $\vec{A}$,$\vec{B}$,$\vec{C}$ conjuntos de tamaño l de vectores, donde l es la cantidad de compuertas del circuito aritmético y $\vec{k}$ un vector con las variables de dicho circuito. Entonces, el formato R1CS del circuito aritmético es:

$$< A, k > * < B, k > - < C, k >= 0$$

Del proceso de "flattening" anterior, se pueden extraer las variables {*x, y, aux_1, aux_2, out*}. Con estas variables se construye el vector $k = [1, x, y, aux_1, aux_2, \ out]$. Notar que para cualquier R1CS, el primer miembro del vector k es un "1", como es necesario según se explica en el teorema 12 de la sección 7.3 de [32].

Entonces, para la primera compuerta se tiene que:

$$A_{i=1} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \ B_{i=1} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \ C_{i=1} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Luego

$$< A_1, k > * < B_1, k > - < C_1, k >= x * x - y = 0$$

,
es exactamente la primera compuerta. Tomando en consideración todo el circuito aritmético, se llega al sistema R1CS:

$$A = \begin{bmatrix} 0 & 6 & 0 & 13 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix} B = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} C = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Esta es la representación de R1CS de la computación C, donde la solución se encuentra con $x = -3$. Con este valor se puede reconstruir el vector $\vec{k}$ como un vector de solución específico para C, denominado $\vec{k_C}$ sabiendo que:

```
9 = -3 * -3
-18 = 6* -3
-9 = 9+-18
~out = -9+13
```

Con out =4 y el vector $\vec{k_C} = [1, -3, 9, -18, -9, 4]$

Teniendo el R1CS listo, lo único que falta es aplicar interpolador de Lagrange a cada vector y se obtendrán los polinomios para cada compuerta en el formato de la QAP. Recordando que, la interpolación de Lagrange genera un polinomio $P(x)$ de la forma:

$$P_{A1}(x) = \prod_{i=0, i \neq j}^{N} \frac{x - x_i}{x_j - x_i}$$

Donde $x_i, x_j \in A_1$ y N es el tamaño (no magnitud) de A_1. Si se realiza la interpolación de Lagrange a cada vector de A, B y C se puede construir los grupos de polinomios de la ecuación 6.2 para la cual $\vec{k_C}$ corresponde a los valores $\{a_1, a_2, ..., a_n, ..., a_m\}$. El polinomio $t(x)$ de la ecuación 6.2 es construido con los valores $r_g \in M$, corre-

spondientes a los valores escalares de las compuertas de multiplicación de salida del circuito aritmético, donde M es el conjunto de todas las compuertas como se detalla en el ejemplo de la sección 7.4 de [32]:

$$t(x) = (x - r_{g1})(x - r_{g2})...(x - r_{gn})$$

6.1.3 Conocimiento del Exponente

Habiendo formulado el problema en el formato de un QAP se pueden empezar a crear los objetos que compondrán la ZK-SNARK. Pero para entender cómo, por qué y para qué se construyen estos objetos se debe entender la matemática que los sustenta.

Se vio que un QAP presenta un mecanismo para verificar el conocimiento de los testigos, pero que dicho mecanismo no es ni sucinto ni de cero conocimiento, ya que los valores $\vec{a} = (a_1, a_2, ...a_n, ...a_m)$, que resuelven el programa $C()$, son expuestos. Para ocultar estos valores, se utiliza una clase especial de aritmética sobre un campo numérico diferente.

Sea $\mathbb{F}$ un cuerpo finito, con $\mathbf{G}$ su generador, y cuyo producto escalar está definido de forma similar al "problema de logaritmo discreto" (DLP, por sus siglas en inglés) descrito en la sección 1.1 de [33]. Para $(P, G) \in \mathbb{F}$ y un entero x, conocido, se puede calcular $P = x*G$ con un algoritmo en tiempo polinomial. Pero no existe un algoritmo con tiempo polinomial que pueda encontrar un entero y, tal que $P = y * G$ [33]. Entonces, si el par (G, P) son números suficientemente grandes, recuperar y para alguien que no conoce su valor a través de (P, G) es computacionalmente inviable.

De esta propiedad se puede derivar otra aún más importante y central en las ZK-SNARK, el supuesto de "Conocimiento del Exponente" (KEA1 por sus siglas en inglés *Knowledge of Exponent Assumption*), introducido en [31]. Sean $(H, J) \in \mathbb{F}$. Sea un valor $P = p * H$ dado. Una entidad E no puede generar otro valor $Q = p * J$ sin el conocimiento de p a menos que $(Q, J) = (r * P, r * H)$, para algún valor r conocido por E [38]. No podría, por ejemplo, generar un punto $Q' = p * K$ para un $K \in \mathbb{F}$, $K \neq H \wedge K \neq J$.

Además sea e una función llamada "función de bilinealidad" definida como [39]: $e : G_1 \times G_2 \rightarrow G_T$ con G_1, G_2 y G_T grupos cíclicos en $\mathbb{F}$ que satisface:

$$e(x, y + z) = e(x, y) * e(x, z)$$

$$e(x + y, z) = e(x, z) * e(y, z)$$

$$e(x, ay) = a * e(x, y) = e(ax, y)$$

Con esto otras entidades pueden verificar que el par (Q, J) generado por la entidad E mediante el escalar r sin el conocimiento ni del escalar k ni de r a través de[38]:

$$e(J, P) = e(H, Q)$$

Ya que si $P = p * H$ y $(Q, J) = (r * P, r * H)$ entonces:

$$e(J, P) = e(H, Q)$$

$$e(r * H, P) = e(H, r * P)$$

$$r * e(H, P) = r * e(H, P)$$

Esto se puede extender a más puntos. Sean los pares $(P_1, H_1), (P_2, H_2)...(P_n, H_n)$ que cumplen con $P_i = p*H_i$, $i \in \{1, 2, ..., n\}$. Entonces una entidad E que no conoce el valor p puede crear un par $(R, S) := R = k * S$ donde (R, S) son una combinación lineal de los valores (P_i, H_i) de la forma $S = H_1 * l_1 + H_2 * l_2 + ... + H_n * l_n$ y $R = P_1 * l_1 + P_2 * l_2 + ... + P_n * l_n$ donde los valores $l_1, l_2, ..., l_n$ son conocidos por E.

6.1.4 Curvas Elípticas

Hasta el momento se ha hablado de un campo $\mathbb{F}$ que sostiene propiedades importantes para las ZK-SNARKs, pero no se ha entrado en el detalle. Un grupo que sostiene estas propiedades, además de otras interesantes para la criptografía, es el grupo cíclico

definido por una curva elíptica sobre un campo finito $\mathbb{F}_p$, donde p es un número primo *grande*.

Una curva elíptica, como la que se muestra en la figura 6.1, es una curva formada por la ecuación:

$$y^2 = x^3 + bx + c \leftrightarrow 4b^3 + 27c^3 \neq 0 \tag{6.3}$$

Esta es una curva elíptica en la forma de Weierstrass [33].

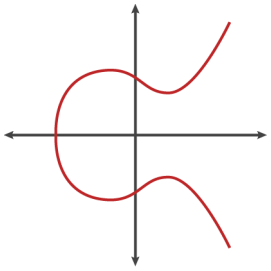


Fig. 6.1: Curva elíptica formada por la ecuación $y^2 = x^3 - x + 1$.

Entonces, se puede definir el conjunto **F** de puntos en una curva elíptica con parámetros b y c, sobre un campo finito $\mathbb{F}_p$ con un número primo p:

$$\{P = (x, y) \in \mathbb{F}_p : y^2 = x^3 - bx - c \ (mod\ p)\}$$

Notar que los elementos de **F** son puntos en una curva y no simples números. Este grupo cuenta con un punto **G** como su generador.

En la sección 2.1.1 de [33] se define la operación de adición, que es la norma del grupo **F**, como sigue:

Sean $P = (x_1, y_1)$ y $Q = (x_2, y_2)$ puntos arbitrarios de la curva elíptica E, en la forma de Weierstrass. La operación $P + Q = R = (x_3, y_3)$ es:

$$x_3 = \lambda^2 - x_1 - x_2\,,$$

$$y_3 = \lambda(x_1 - x_3) - y_1\,,$$

donde:

$$\lambda = \frac{y_1 - y_2}{x_1 - x_2}$$

Si $P = Q$ entonces $P + Q = 2P$, esta operación es conocida como duplicación y se define como:

$$x_3 = \lambda^2 - 2x_1 \,,$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \,,$$

donde:

$$\lambda = \frac{3x_1^2 + b}{2y_1}$$

Por último, el cálculo de $Q = n * P$ con n un entero positivo -cualquiera-, es realizar n duplicaciones sobre P.

$$n * P = \underbrace{P + P + P... + P}_{\text{n veces}} = (n-1) * P + P$$

Este proceso se puede llevar a cabo de distintas formas, con algoritmos y formas de curva distintas. La sección 2.1.4 de [33] demuestra que este proceso es un algoritmo de tiempo polinómico y la tabla 2.1, de la sección 2.5 de [33], muestra la comparación de los costos de la duplicación en distintas formas de la curva.

Además, en curvas elípticas se puede definir un problema similar al DLP, llamado ECDLP. Sean $(P, Q) \in E(\mathbb{F}_p)$ con E una curva elíptica, como la de la ecuación 6.3. El problema de encontrar un entero k, tal que $P = k * Q$, es llamado ECDLP ("*Elliptic Curve Discrete Logarithm Problem*"). Actualmente no existe algoritmo de complejidad sub-exponencial que pueda resolver este problema [33].

El Lemma III.9 de [34] define el "emparejamiento de Weil" en un grupo definido por una curva elíptica y el Lemma III.10 muestra que dicho emparejamiento cumple con ser una función bilineal. Por lo tanto, una función e, como la definida en la sección 6.1.3, existe para las curvas elípticas. Así, todas las operaciones, definidas en

la sección 6.1.3, se pueden aplicar a un grupo $\mathbf{F}$ definido por los puntos pertenecientes a una curva elíptica E sobre un campo primo $\mathbb{F}_p$.

6.1.5 Evaluación ciega de un QAP

Con una QAP, que describe un programa $C()$, se puede verificar el conocimiento de los testigos, pero dicha verificación los revela, al ser una combinación lineal entre ellos y los polinomios V, W, Y, t, como se vio en la ecuación 6.1. Sin embargo, con lo aprendido en la sección 6.1.3 se puede verificar, mediante la función e, la relación que existe entre dos pares (P, Q) y (R, S), sin revelar como se construyen estos valores. El problema es que esta verificación funciona sólo para puntos $P, Q, R, S \in \mathbb{F}$ que cumplan estas propiedades y no para los polinomios de la QAP.

Entonces, se procede a convertir los polinomios en elementos del campo $\mathbb{F}$:

Sean $A(x)$ un polinomio cualquiera. Sean τ, ρ y α, números enteros aleatorios y $\mathbf{G}$, un punto de una curva elíptica E sobre un campo primo $\mathbb{F}_p$. Entonces, se puede crear los puntos.

$$
\begin{aligned}
Q :=& \rho * \mathbf{G} \\
Q' :=& \alpha * \rho * \mathbf{G} \\
P :=& \rho * A(\tau) * \mathbf{G} = A(\tau) * Q \\
P' :=& \alpha * \rho * A(\tau) * \mathbf{G} = A(\tau) * Q'
\end{aligned}
$$

Lo cual cumple con la propiedad de *KEA1* de la sección 6.1.3, ya que $(P, P') = (r * Q, r * Q')$ para un par (Q, Q') ; $Q' = \alpha * Q$ donde $r = A(\tau)$. Por lo tanto, se puede transformar un polinomio $A(x)$ cualquiera a un punto $P \in \mathbb{F}_p$ que esconde todo lo relacionado a él y se puede usar para la prueba del KEA1.

6.2 Pinocchio: Algoritmo de Computación Verificable

El esquema de verificación de las ZK-SNARK utiliza un algoritmo de verificación de 3 pasos, detallado en la sección 2.1 de [35]:

- **Generación de llaves**: Proceso de inicialización de los elementos a utilizar en el algoritmo. Se crean dos valores pk y vk que son utilizados para generar y evaluar pruebas.
- **Proceso de creación de Pruebas**: Proceso en que se crean los valores π_y y $\vec{x}$, que son la prueba y los valores públicos que la prueba verifica, respectivamente.
- **Verificación de pruebas** Proceso final en donde se verifica la prueba y se produce un valor de decisión.

6.2.1 Generación de llaves

Una ZKP verifica la ecuación 5.1. De esta ecuación se obtiene el QAP de la computación $C()$. Sean

$$V(x) = \{V_1(x), V_2(x), ..., V_m(x)\}$$

$$W(x) = \{W_1(x), W_2(x), ..., W_m(x)\}$$

$$Y(x) = \{Y_1(x), Y_2(x), ..., Y_m(x)\}$$

, los conjuntos de polinomios que satisfacen la ecuación 6.1 y que corresponden al QAP de la computación $C()$. Sean los valores τ, ρ_1, ρ_2, α_1, α_2, α_3 y β, números enteros aleatorios desconocidos por cualquier entidad. Sean los grupos cíclicos $\mathbf{F_1}$, $\mathbf{F_2}$, definidos por una curva elíptica sobre un campo primo $\mathbb{F}_p$ y $\mathbf{G_1}, \mathbf{G_1}$, sus respectivos generadores. Se definen los valores como se establece en la figura 10 de [30] y la

sección 3.3 y 3.4 de [36]:

$$pk_V := \{V_i(\tau)\rho_1 * \mathbf{G_1}\}_{i=1}^m \qquad pk'_V := \{\alpha_1 V_i(\tau)\rho_1 * \mathbf{G_1}\}_{i=1}^m$$
$$pk_W := \{W_i(\tau)\rho_2 * \mathbf{G_2}\}_{i=1}^m \qquad pk'_W := \{\alpha_2 W_i(\tau)\rho_2 * \mathbf{G_1}\}_{i=1}^m$$
$$pk_Y := \{Y_i(\tau)\rho_1\rho_2 * \mathbf{G_1}\}_{i=1}^m \qquad pk'_Y := \{\alpha_3 Y_i(\tau)\rho_1\rho_2 * \mathbf{G_1}\}_{i=1}^m$$

$$pk_K := \{\beta(V_i(\tau)\rho_1 + W_i(\tau)\rho_2 + Y_i(\tau)\rho_1\rho_2) * \mathbf{G_1}\}_{i=1}^m$$

$$pk_H := \{\tau^i * \mathbf{G_1}\}_{i=0}^d$$

Estos valores conforman $\mathbf{pk} = \{pk_V, pk_W, pk_Y, pk'_V, pk'_W, pk'_Y, pk_K, pk_H\}$, correspondiente a la llave que se ocupa para generar las pruebas, denominada la "llave del proveedor".

A los valores τ,ρ_1,ρ_2,α_1,α_2, α_3 y β se le agrega un valor extra, γ, y con estos se calculan:

$$vk_V := \alpha_1 * \mathbf{G_2} \qquad vk_W := \alpha_2 * \mathbf{G_1} \qquad vk_Y := \alpha_3 * \mathbf{G_2}$$
$$vk_\gamma := \gamma * \mathbf{G_2} \qquad vk_{\beta\gamma 1} := \beta\gamma * \mathbf{G_1} \qquad vk_{\beta\gamma 2} := \beta\gamma * \mathbf{G_2}$$

$$vk_t := t(\tau)\rho_1\rho_2 * \mathbf{G_2} \qquad vk_{IC} = (V_i(\tau)\rho_1 * \mathbf{G_1})_i^n = 0$$

Estos valores conforman $\mathbf{vk} = \{vk_V, vk_W, vk_Y, vk_\gamma, vk_{\beta\gamma 1}, vk_{\beta\gamma 2}, vk_t, ck_{IC}\}$, conocida como "llave de verificación".

Ambos valores $(\mathbf{pk}, \mathbf{vk})$ son parte de la "*Common Reference String*" (CRS) del programa $C()$, correspondiente a los valores públicos necesarios para crear y verificar pruebas ZK-SNARKs para el programa $C()$.

6.2.2 Creación de la prueba

El emisor de la prueba, quien conoce los valores $\{a_1, a_2, ..., a_n, ..., a_m\}$, que incluye los valores $(\vec{Y}, \vec{x}, \vec{z})$ del esquema $C(\vec{x}, \vec{z}) = \vec{Y}$, computa el QAP Q para el programa

$C()$ con los grupos de polinomios V, W, Y y el polinomio $t(x)$, según se vio en las secciones 6.1.1 y 6.1.2. Calcula el vector k_C definido en la sección 6.1.2 y define los números aleatorios $(\delta_1, \delta_2, \delta_3)$ de su conocimiento.

El emisor calcula los coeficientes $\vec{h} = [h_1, h_2, .., h_d]$ de $H(x)$ mediante $H(x) = \frac{V(x)W(x)-Y(x)}{t(x)}$, donde:

$$V(x) := V_0(x) + \sum_{i=1}^{m} a_i V_i(x) + \delta_1 t(x)$$
$$W(x) := W_0(x) + \sum_{i=1}^{m} a_i W_i(x) + \delta_2 t(x)$$
$$Y(x) := Y_0(x) + \sum_{i=1}^{m} a_i Y_i(x) + \delta_3 t(x)$$

Con esto y los valores de la CRS, genera los valores según se indica en la figura 10 de [30] y la sección 3.3 y 3.4 de [36]:

$$\pi_V := \sum_{i=n+1}^{m} a_i * pk_{V,i} \qquad \pi'_V := \sum_{i=n+1}^{m} a_i * pk'_{V,i}$$
$$\pi_W := pk_{W,0} + \sum_{i=1}^{m} a_i * pk_{W,i} \qquad \pi'_W := pk'_{W,0} + \sum_{i=1}^{m} a_i * pk'_{W,i}$$
$$\pi_Y := pk_{Y,0} + \sum_{i=1}^{m} a_i * pk_{Y,i} \qquad \pi'_Y := pk_{Y,0} + \sum_{i=1}^{m} a_i * pk'_{Y,i}$$

$$\pi_H := pk_{H,0} + \sum_{i=1}^{d} h_i * pk_{H,i}$$

$$\pi_K := pk_{K,0} + \sum_{i=1}^{m} a_i * pk_{K,i}$$

El emisor. entonces, crea la prueba $\pi := (\pi_V, \pi_W, \pi_Y, \pi'_V, \pi'_W, \pi'_Y, \pi_H, \pi_K)$, públicamente verificable mediante la CRS.

6.2.3 Verificación de la prueba

Un verificador cualquiera recibe la prueba π y las entradas públicas $\vec{x}$ de $C()$, junto a la CRS calcula $vk_x := vk_{IC,0} + \sum_{i=1}^{n} x_i * vk_{IC,i}$ con $x_i \in \vec{x}$. Notar que $\pi_V + vk_x = \vec{x} * V(\tau)\rho_1 * \mathbf{G_1}$. Sea e una función de bilinealidad, como la definida en la sección 6.1.3, donde los grupos G_1 y G_2 son los grupos $\mathbf{F_1}$ y $\mathbf{F_2}$, definidos en la sección 6.2.1. Entonces, el verificador puede comprobar la prueba realizando las siguientes operaciones, según se indica en la figura 10 de [30] y la sección 3.3 y 3.4 de [36]:

- Divisibilidad: $e(vk_y + \pi_V, \pi_W) = e(\pi_H, vk_t)e(\pi_Y, \mathbf{G_2})$
- Extensión: $e(\pi_V, vk_V) = e(\pi'_V, \mathbf{G_2})$, $e(\pi_W, vk_W) = e(\pi'_W, \mathbf{G_2})$, $e(\pi_Y, vk_Y) = e(\pi'_Y, \mathbf{G_2})$
- Uso de los mismos coeficientes : $e(\pi_K, vk_\gamma) = e(vk_y + \pi_V + \pi_Y, vk_{\beta\gamma 2})e(vk_{\beta\gamma 1}, \pi_W)$

Si estas igualdades se sostienen, la prueba π queda verificada. Estas verificaciones se basan en las propiedades de la función e y del supuesto de KEA1 visto en la sección 6.1.3.

6.3 Esquema de funcionamiento y notas sobre ZK-SNARK

Una vez discutido el funcionamiento de Pinocchio, se puede detallar un esquema de funcionamiento general. Este esquema se aprecia en la figura 6.2 y se puede ver un resumen en [36] y en la figura 10 del anexo B de [30].

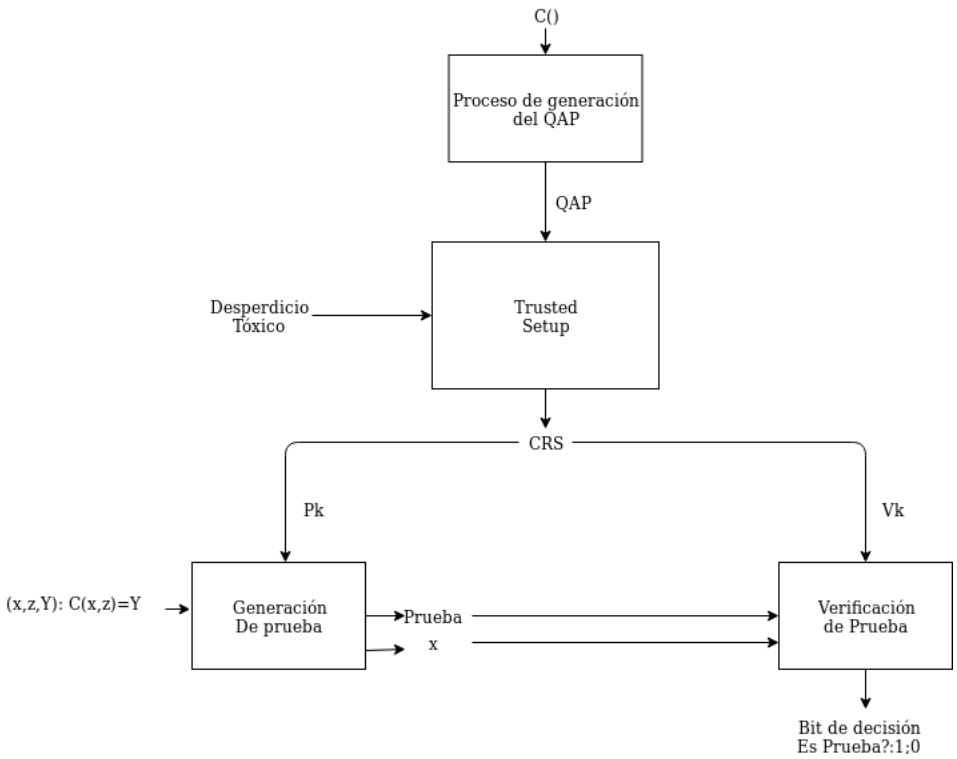


Fig. 6.2: Esquema de funcionamiento de Pinocchio, (x, z) son las entradas de un programa $C()$ que cumple con $C(x, z) = Y$. $CSR = (pk, zk)$ son las llaves del proveedor y la llave de verificación, respectivamente. El desperdicio tóxico corresponde a los valores aleatorios que se utilizan para crear pk y zk, los cuales deben ser destruidos una vez utilizados.

En el contexto de varias entidades que quieren comunicar ZKPs, estas consensúan una computación que cumple $C(\vec{x}, \vec{z}) = Y$, donde $\vec{x}$ son entradas públicas desprotegidas y $\vec{z}$ son los testigos, es decir, valores que se quieren dejar en privado, pero demostrando su conocimiento. Mediante $C()$ se crea un QAP de los cuales se obtienen los grupos de polinomio V, W, Y y un polinomio t. Con el QAP y distintos valores aleatorios se generan llaves de creación y verificación de pruebas. Un emisor de pruebas toma la llave de creación, las entradas $\vec{x}, \vec{z}$ y las salidas $\vec{Y}$ y con ellas genera la prueba π. Por último, cualquier verificador puede comprobar la prueba mediante la llave de verificación vk, la prueba y las entradas públicas $\vec{x}$.

6.3.1 Notas sobre verificación

En la divisibilidad de la etapa de verificación de la ZK-SNARK, la función e actúa de la manera:

$$e(vk_y + \pi_V, \pi_W) = e(\pi_H, vk_t)e(\pi_Y, \mathbf{G_2})$$

$$< \vec{a}, \{V(\tau\}_{i=1}^{m}) > * < \vec{a}, W(\tau) > \rho_1 * \rho_2 * e(\mathbf{G_1}, \mathbf{G_2}) =$$
$$< \vec{h}, H(\tau) > *t(\tau) + < \vec{a}, Y(\tau) >)\rho_1\rho_2 * e(\mathbf{G_1}, \mathbf{G_2})$$

La cual, es cierta si:

$$< \vec{a}, V(\tau) > * < \vec{a}, W(\tau) > = < h_i, H(\tau) > *t(\tau) + < \vec{a}, Y(\tau) >)$$

Con $< a, b >$ producto interno entre a y b, y $\vec{a} = (a_1, a_2, ...a_n, ...a_m)$. Si esta igualdad se mantiene, entonces, se prueba la condición del QAP [37, 32, 30]. Es decir, que $\vec{a}$ cumple la ecuación 6.1 y corresponde al vector $\vec{k_C}$, descrito en la sección 6.1.2. Nótese que, gracias a que la multiplicación escalar con un generador $\mathbf{G} \in \mathbf{F}$ es un ECDLP, como se describe en la sección 6.1.4 (donde $\mathbf{F}$ es un grupo cíclico de una curva elíptica sobre un campo primo $\mathbb{F}_p$), por lo que los valores $\vec{a}$, que incluyen a los testigos $\vec{z}$ de la computación $C()$, quedan ocultos si $\mathbf{G}$ y el número primo p son grandes.

Un atacante podría crear una prueba falsa, sin el conocimiento del vector $\vec{a}$, al proveer los grupos de polinomios $V(x) = 1, W(x) = t(x), Y(x) = 0$ como se plantea en la sección 3.4 de [36]. Es por esto que se agregan los pasos adicionales de ”Extensión” y ”Uso de mismos coeficientes” en el algoritmo de Pinocchio. En el paso de extensión se observa que la igualdad prueba que:

$$e(\pi_V, vk_V) = e(\pi'_V, \mathbf{G_2})$$

$$e(\pi_V, \alpha_1 * \mathbf{G_2}) = e(\alpha * \pi_V, \mathbf{G_2})$$

$$\alpha * e(\pi_V, \mathbf{G_2}) = \alpha * e(\pi_V, \mathbf{G_2})$$

Así, si los polinomios usados por el atacante no son combinaciones lineales de los vectores creados en el R1CS, la igualdad no se mantiene. Este desarrollo se extiende para los grupos $W(x), Y(x)$.

Por otra parte, supóngase que se utilizan los vectores $\vec{b}, \vec{c}, \vec{d}$ en vez de $\vec{a}$ para crear las pruebas (π_V, π_W, π_Y). Entonces, se comprueba que:

$$e(\pi_K, vk_\gamma) = e(vk_y + \pi_V + \pi_Y, vk_{\beta\gamma 2})e(vk_{\beta\gamma 1}, \pi_W)$$

$$e(\pi_K, \gamma * \mathbf{G_2}) = e(vk_y + \pi_V + \pi_Y, \beta\gamma\mathbf{G_2})e(\beta\gamma * G_1, \pi_W)$$

$$e(< \vec{a}, \beta\rho_1(V(\tau)) > *\mathbf{G_1}, \gamma * \mathbf{G_2}) * e(< \vec{a}, \beta\rho_2(W(\tau)) > *\mathbf{G_1}, \gamma * \mathbf{G_2}) *$$
$$e(< \vec{a}, \beta\rho_1\rho_2(Y(\tau)) > *\mathbf{G_1}, \gamma * \mathbf{G_2}) =$$
$$e(< \vec{b}, \rho_1(V(\tau)) > *\mathbf{G_1}, \beta\gamma * \mathbf{G_2}) * e(< \vec{c}, \rho_2(W(\tau)) > *\mathbf{G_1}, \beta\gamma * \mathbf{G_2}) *$$
$$e(< \vec{d}, \rho_1\rho_2(Y(\tau)) > *\mathbf{G_1}, \beta\gamma * \mathbf{G_2})$$

$$\beta^3\gamma^3\rho_1^2\rho_2^2 < \vec{a}, (V(\tau)) >< \vec{a}, (W(\tau)) >< \vec{a}, (Y(\tau)) > *e(\mathbf{G_1}, \mathbf{G_2}) =$$
$$\beta^3\gamma^3\rho_1^2\rho_2^2 < \vec{b}, (V(\tau)) >< \vec{c}, (W(\tau)) >< \vec{d}, (Y(\tau)) > *e(\mathbf{G_1}, \mathbf{G_2})$$

La igualdad se mantiene solo si $\vec{b} = \vec{c} = \vec{d} = \vec{a}$, como se establece en la definición 12 de la sección 7.1 de [32].

6.3.2 Trusted Setup

Uno de los mayores problemas de las ZK-SNARKs es el proceso llamado ”trusted setup”, que corresponde a la etapa de generación de llaves del algoritmo de Pinocchio.

En esta etapa, explicada en la sección 6.2.1, los valores $\Phi_{TW} = (\tau, \rho_1, \rho_2, \alpha_1, \alpha_2, \alpha_3, \beta, \gamma)$ se usan para generar las llaves (pk, vk) de la CRS. La prueba π se basa en que es el emisor el único que puede generarla, ya que debe cumplir el supuesto del KEA1 para ocupar las llaves, porque nadie conoce el valor de los valores aleatorios. Pero, si por alguna razón un atacante supiera estos valores, podría generar pruebas falsas al ser capaz de encontrar puntos $\pi_\phi = \delta\phi * \mathbf{G_i}$ con $\phi \in \Phi_{TW}$ y donde δ es un valor que cumple con las ecuaciones de verificación.

Esto es el equivalente a que A conozca la fuente de aleatoriedad de B, en el ejemplo de la sección 5.3.1. Es por esto que a Φ_{TW} se le conoce como "Desecho Tóxico" (*Toxic Waste*) [38, 42] como aparece en la imagen 6.2. Este desecho tóxico debe ser eliminado una vez generada las llaves, para evitar cualquier ataque al sistema que ocupe el programa $C()$. ZCash, una criptomoneda, cuyo sistema de transacción se sustenta en las ZK-SNARKs, realizó una ceremonia para generar estos parámetros con distintos participantes [42]. Hasta ahora, esta es una de sus mayores fuente de críticas, al no ser 100% comprobable que nadie conoce dichos parámetros.

6.4 Groth 2016, Algoritmo eficiente de pruebas ZK-SNARK

Con Pinocchio se comenzó a hablar de ZK-SNARK, pero actualmente se utilizan 3 tipos de algoritmos, siendo Pinocchio el primero de ellos. El último de estos algoritmos es Groth16, ideado por Jens Groth en [40]. Con este algoritmo Groth logra reducir el tamaño de la prueba de los 8 elementos de Pinocchio a solo 3.

6.4.1 El algoritmo

Groth mantiene la base algorítmica de Pinocchio. En el algoritmo aún usa las QAP, los grupos bilineales y los 3 procesos: de inicialización, generación y verificación. Además de agregar un 4to proceso de simulación para demostrar la propiedad de cero conocimiento perfecto, detallada en la definición 2 de la sección 2.2 de [40].

Para lograr esto, el algoritmo de Groth divide la CRS en 2 vectores σ, creados en

un proceso de generación, mediante los elementos $\alpha, \beta, \gamma, \delta, \tau$, tal como se define en la sección 3.2 de [40]:

Sea R una QAP como la definida en la sección 6.2.1 y $\alpha, \beta, \gamma, \delta, \tau$, números generados aleatoriamente. Entonces,

$$\sigma \leftarrow Setup(R, \alpha, \beta, \gamma, \delta, \tau)$$

$$\sigma = (\sigma_{\mathbf{1}}, \sigma_{\mathbf{2}})$$

Donde $\sigma_{\mathbf{1}}, \sigma_{\mathbf{2}}$ están definidos en la sección 3.2 de [40]. Para la fase de generación de prueba, el emisor de la prueba elige dos valores aleatorios (r, s) y genera la prueba $\pi = ([A]_1, [B]_2, [C]_1)$, donde el operador $[.]_i$ se define como $[k]_i = k * \mathbf{G}_i$, con $\mathbf{G}_i$ un generador de un grupo bilineal como los definidos en 6.2.1. Por último, el proceso de verificación consiste en una ecuación que verifica el emparejamiento de la prueba con la CRS:

$$\sigma_{x1} = \left\langle \{a_i\}_{i=0}^{n}, \left\{ \frac{\beta U(\tau) + \alpha V(\tau) + W(\tau)}{\gamma} \right\}_{i=0}^{n} \right\rangle$$

$$[A]_1 * [B]_2 = [\alpha]_1 * [\beta]_2 + [\sigma_{x1}]_1 * [\gamma]_2 + [C]_1 * [\delta]_2 \tag{6.4}$$

Esta igualdad se mantiene si $\vec{a} = (a_1, a_2 ... a_n, .. a_m)$ son tales que cumplen con el planteamiento de la QAP, definida en la sección 6.1.1. El Teorema 1 y 2 de [40] comprueba que el algoritmo de Groth cumple con completitud, cero conocimiento perfecto y robustez perfecta. La función de los elementos α y β es de verificar que los 3 elementos de π utilizan el mismo vector $\vec{a}$, para su generación de forma similar como lo hace la ecuación 6.4. Los valores (r, s) generan una prueba aleatoria que permite el cero conocimiento, de forma similar a los valores $(\delta_1, \delta_2, \delta_3)$ definidos por el emisor en el protocolo de Pinocchio de la sección 6.2.2.

6.4.2 Maleabilidad de Groth16

El protocolo de Groth16 tiene la propiedad de ser maleable, es decir, un atacante puede generar una prueba válida sin conocer los testigos de la computación $C()$. Este

proceso es descrito en [41].

Para prevenir este ataque se debe generar un registro que anule cualquier prueba que ocupe esta prueba. Una técnica utilizada para este fin, es la disposición de un ”nulificador”, el cual es un elemento generado a partir de los datos ocupados para crear la prueba. Es importante que este nulificador no pueda revelar ninguna información de los datos con la que fue creado. El nulificador es publicado y anula cualquier otra prueba que utilice los mismos datos de la prueba a la cual pertenece, con esto se previene un ataque por maleabildiad. Una descripción más detallada del nulificador se revisará en la sección 7.3.

7. PROGRAMA DEL VOTO

En el capítulo 5 se discutió la identidad digital, cómo idear las credenciales de los votantes y cómo comprobar su identidad, mientras que en el capítulo 6 se presentó un esquema de cero conocimiento para ocultar las credenciales de los votantes. Para esto últimos se requiere construir un programa $C()$ que cumpla con lo discutido en ambos capítulo.

7.1 Restricciones y propiedades del programa

El programa del voto está limitado por distintas fuerzas dentro de la implementación: Por los objetivos propios del sistema, las limitaciones propias de Hyperledger Fabric y las limitaciones de ZK-SNARK. Sin embargo, estas directrices también proveen al programa de propiedades especiales.

Según los objetivos del sistema, un voto debe cumplir con:

- Proveer una tendencia clara: El formato es homogéneo y la elección puede identificarse y contarse.
- Ser auténtico: Se puede verificar que el voto proviene de su autor original.
- Ser verídico: Se puede verificar que su autor está habilitado para votar.
- Ser robusto: El valor del voto mantiene la intención original del votante al momento de contarse.
- Ser anónimo: No se puede vincular un voto con su autor.

La primera propiedad se puede lograr simplemente con un campo en el voto que muestre un valor estandarizado de las opciones a elegir. El resto de las propiedades se puede cumplir mediante una ZK-SNARK que demuestre que el votante está habilitado para votar, pero que no revele sus credenciales.

En cuanto a la propiedad de ser verídico, se debe demostrar que el votante está inscrito en el padrón electoral, que es quien dice ser y que es la primera vez que vota. Las ZK-SNARK no revelan nada de los testigos. Es por esto que, las dos pruebas, π y π', que ocupan los mismos testigos, entradas pública y programa $(\vec{z}, \vec{x}, C())$, cumplen con $\pi \neq \pi'$ para evitar revelar que los mismos testigos fueron usados en su creación. Esto dificulta la tarea de verificar que el votante no ha votado más de una vez, ya que no hay nada en el voto que lo indique. Se debe agregar un tercer valor, que se repita cuando los mismos testigos sean utilizados sin revelarlos.

Por lo tanto, un voto $\mathbf{v}$ está compuesto por tres valores, el valor de la elección, una prueba ZK-SNARK y un nullifier que evidencia cuando los testigos son ocupados de nuevo.

7.2 Identificación de los votantes

En el capítulo 5 se habló de cómo se podía ocupar un par de secreto s e imagen S que cumplen con $H(s) == S$ para identificar a los votantes, con H una función "Hash". Pero se necesita un mecanismo para identificar si dichas credenciales están habilitadas para votar. El mecanismo debe tomar la imagen S y encontrarla en un padrón electoral. Si S pertenece al padrón, entonces se demuestra que el votante está inscrito.

Sin embargo, no se puede consultar una lista externa a la prueba puesto que se revelaría la identidad del votante. Pero si se puede demostrar que S pertenece a un cierta porción del padrón electoral. Esta porción estaría incluida en la prueba, ya que revelarla puede dar información respecto a la llave. Esto puede lograrse a través de una lista.

7.2.1 Verificación de habilidad

Con todo, se puede generar un padrón electoral compuesto de listas de los ”Hash” del secreto de los votantes. Notar que no se necesita una lista única ni un tamaño en específico, sino que debe ser de un tamaño lo suficientemente grande para garantizar el anonimato de un votante individual.

Cada votante V, para la creación de la prueba, debe presentar su secreto s y un subconjunto P_V del Padrón electoral..

Así, la computación $C()$ para la creación de la ZK-SNARK, se define como:

```
C(s,P_V){
    S := H(s)
    idx := 1
    for p in P_V{
        idx *= S-p   // (S-p1)(S-p2)...(S-pV)
    }
    return idx == 0
}
```

Listing 7.1: Programa para la verificación de la habilitación del votante V

7.3 Nullifier y Lógica de verificación

Se discutió en la sección 7.1 que se necesita un valor que se repita cada vez que se utilicen los testigos, pero que no los revele. En la sección 1.2 de [46] se introduce el concepto de ”número serial”, que posteriormente ZCash [45] rebautiza como ”nullifier” (nulificador).

7.3.1 Nullifier

El número serial ”sn_T” corresponde a una transacción T oculta y verificada por una ZKP π y es conocida únicamente por el emisor E_T de la transacción. Luego, el emisor E_T usa un valor auxiliar r para crear el valor $h_{snT} = H(sn_T||r)$, con H una función

Hash resistente a colisiones. Posteriormente h_{snT} es publicado en la red, la cual lo guarda con el estado de ”disponible” (como un UTXO, definido en [10]). cuando E_T envía la transacción T a un receptor R_T, es decir, le ”paga” con la transacción T, le revela el valor sn e incluye en π una declaración que dice ”existe un valor r, tal que $h_{snT} = H(sn_T||r)$, donde h_{snT} está publicado en la red”. Con esto E_T le cede el derecho a R_T de ocupar la transacción T.

R_T al crear una transacción T', que incluya a T, debe emitir una prueba que declare conocer el valor sn_T. Con esto la red cambia el estado de h_{snT} a ”gastado” y, luego, puede verificar si R_T u otra entidad quiere volver a ocupar sn, así, evitando el problema del ”doble gasto” manteniendo el anonimato de E_T.

Este concepto puede ser ocupado en el sistema de votación, al obligar al votante V a incluir en la prueba π una declaración que diga ”El votante conoce un valor $sn = From(S)$, tal que $h_{sn} = H(sn)$, donde h_{sn} *no* ha sido publicado en la red anteriormente”. Con $From()$ una función cualquiera que cree sn, usando la imagen pública S. El pedir que h_{sn} no haya sido publicada antes significa que es la primera vez que se utiliza la credencial sn y, por lo tanto, S. Es decir, es la primera vez que el votante V vota. Si el votante intenta votar de nuevo, volverá a ocupar sn por lo que la red podrá darse cuenta que intenta votar más de una vez.

7.3.2 Lógica final

Consecuentemente, en el voto se debe incluir el valor h_{sn} y en la prueba se debe agregar este valor sn y demostrar que el h_{sn} publicado corresponde al sn. Para estos efectos, se puede ocupar ambos valores s y S del voto, ya que sn debe ser derivado de S. En este caso s se utiliza para ofuscar el valor sn y no se pueda deducir la identidad del votante una vez publicado, ya que s es secreto en todo momento. Así, el programa $C()$ final queda:

```
1 C(s,P_V, h_sn){
2     S := H(s)
3     idx := 1
```

```
    for p in P_V{
        idx *= S-p   // (S-p1)(S-p2)...(S-pV)
    }
    isSn = H(s||S)
    return idx == 0 && isSn == h_sn
}
```

Listing 7.2: Programa para la verificación de la habilitación del votante V, similar a 7.1, pero que incluye la verificación del ”nullifier”.

Con esto se puede crear una prueba π mediante el protocolo de Pinocchio:

$$\vec{x} = (P_V, h_{sn})$$
$$\vec{z} = (s)$$
$$\vec{Y} = \{0, 1\}$$

Finalmente, un voto es una estructura compuesta por el ”nullifier”, la prueba ZK-SNARK del programa 7.2, donde $C(\vec{x}, \vec{z}) = \vec{Y}$ son sus entradas públicas y el valor del voto es de la forma $Vote = (h_{sn}, (\pi, \vec{x}), m)$, donde este último provee de una tendencia clara mediante m. El voto es autentico, verídico y robusto debido a $h_{sn}, \pi, \vec{x}$ y el anonimato está garantizado, al no revelarse los testigos $\vec{z}$.

8. IMPLEMENTACIÓN DE UN SISTEMA REV-E2E

Como se mencionó en la introducción, el sistema a implementar es una sistema de votación remoto con verificación ”end to end”. Hasta el momento se ha presentado una infraestructura mediante Hyperledger Fabric, que permite la votación a través de consolas de votación. Estas consolas están diseñadas como un software. El hardware que lo ejecuta no es parte del alcance de esta memoria, pero será discutido posteriormente. Paralelamente se presentó una lógica de voto que permite el anonimato y la verificación, simultáneamente. A continuación se presentará el software necesario para incorporar dicha lógica con la infraestructura de Hyperledger Fabric.

8.1 ZoKrates

El proceso de convertir un programa a una ZK-SNARK es extenso y tiene distintas limitaciones matemáticas. Por lo que para construir una QAP desde el programa 7.2, no es viable de hacer manualmente, debido a la cantidad de compuertas aritméticas que requiere la implementación de firmas digitales y funciones Hash, necesarias para su programación. Es por esto que se requiere una herramienta que provea la plataforma para construir el programa, realizar el ”trusted setup”, generar pruebas y verificarlas.

Para estos efectos se eligió el uso de ZoKrates. ZoKrates es un grupo de herramientas que ofrece la construcción de pruebas ZKP mediante ZK-SNARK para su implementación ”fuera de cadena” [53]. ZoKrates cuenta con un lenguaje de dominio específico (DSL por sus siglas en inglés) y con un cliente. Mediante el DSL se pueden codificar programas que luego, el cliente, puede convertir en esquemas ZK-SNARK [53].

8.1.1 DSL

Debido a la complejidad de ZK-SNARK, el DSL ofrece funcionalidades limitadas. Tiene 3 tipos de datos distintos: ”Field”, que es un tipo de dato entero positivo que puede llegar hasta un número primo de 254 bits [53]. Arreglos de ”Field”, que se comportan como una lista indexada de Fields y datos ”booleanos” que sólo pueden ser evaluados y no pueden ser ingresados ni retornados por una función.

En cuanto a los flujos, el DSL sólo permite expresiones ”if” y ”for”, las funciones deben definirse antes de ser ocupadas y las asignaciones son solo por valor y de forma determinista (todos los arreglos tienen tamaño fijo, por ejemplo). Las entradas a las funciones pueden definirse como ”private”, lo cual convierte a esa entrada en un testigo. Si no se define como ”private”, se asume que es una entrada pública. Un ejemplo de implementación de una función se puede ver a continuación:

```
1 import "hashes/sha256/512bitPacked" as sha256packed
2
3 def main(private field a, private field b, private field c, private
    field d) -> (field):
4     h = sha256packed([a, b, c, d])
5     h[0] == 263561599766550617289250058199814760685
6     h[1] == 65303172752238645975888084098459749904
7     return 1
```

Listing 8.1: Programa del tutorial de ZoKrates. Recibe la pre-imagen de un hash , aplica la función Hash y lo compara con el Hash original. Con esto se pretende probar el conocimiento del Hash de un valor.

8.1.2 Cliente

El cliente es un programa que realiza las operaciones del protocolo de verificación elegido (Groth16 por defecto). Las tareas que realiza están descritas en la sección IV.B de [53], pero de ellas destaca:

- Compilador: Convierte el programa escrito en DSL al formato R1CS.

- Inicialización: Genera las llaves del programa compilado.
- Generador de testigos: Toma los testigos y crea el vector adecuado para la R1CS con ellos.
- Generación de Pruebas: Crea la prueba mediante los testigos y la llave de generación.
- Exportación de llave: Crea un archivo con la llave de verificación y, alternativamente, un contrato inteligente de Ethereum que, internamente, tiene la llave de verificación y puede comprobar la prueba.

8.2 Creación del voto

Según lo visto en el capítulo 7 un votante debe proveer un secreto. Este dato debe ser procesados mediante el DSL de ZoKrates para la creación de la prueba. Por lo que es importante que existan librerías de una función Hash resistente a colisiones disponible en el DSL.

Para ello ZoKrates cuenta con los esquemas de Hash SHA256 y Pedersen. En este trabajo, se utilizó Pedersen debido a su eficiencia en las SNARKs.

A cada votante registra una Imagen $S = H(s)$ que lo identifica públicamente en la cadena. Esta imagen es agregada a una sub-lista de otras identidades que componen el padrón electoral. Al momento de Votar la sub-lista correspondiente es traida desde la red y utilizada para crear el voto. El programa en el DSL de Zokrates para la creación del voto es de la forma:

```
1 import "hashes/pedersen/512bit.zok" as hash
2
3 def main(private u32[8] preImg, u32[10][8] voteRoll, u32[8] voteCast
    , u32[8] votingConst, u32[8] nullifier, u32[8] signCast):
4     u32[8] ans = hash([...preImg, ...preImg])
5     field isIn = 1
6     field comparator = 0
```

```
    for field i in 0..10 do
        comparator = if (ans == voteRoll[i]) then 0 else 1 fi
        isIn = isIn * comparator
    endfor
    assert(isIn == 0)

    u32[8] isNull = hash([...preImg, ...ans])
    assert(isNull == nullifier)

    u32[8] isVote = hash([...voteCast, ...votingConst])
    assert(isVote == signCast)
    return
```

Listing 8.2: Computación C() que verifica la habilidad del votante para sufragar.

Notar los valores voteCast, votingConst y signCast. Estos valores son, respectivamente, el hash del valor del voto, una constante global del sistema traída desde la red y un ”Hash” de ambas combinadas. Esto se utiliza para verificar que el voto emitido se corresponde con la prueba, y evitar que una prueba sea secuestrada por otro votante.

8.3 Chaincode

En el capitulo 4 se describió como un sistema de votación es construido mediante la infraestructura de Hyperledger Fabric. Sin embargo, de los tres softwares descritos solo se profundizó en dos, sin entrar en detalles en el ”chaincode”, esto debido a que no se habían introducido las nociones de anonimato, pero ahora que se conocen las pruebas ZK-SNARK y el software ZoKrates se puede entrar a discutir al construcción del ”chaincode”.

El chaincode presenta distintos métodos los cuales son:

```
{
func (s *SmartContract) CcSetup(.. padron string  ) error
func (s *SmartContract) CcAddId(.. id []string  ) error
func (s *SmartContract) CcGetArtifacts() (string, error)
```

```
5 func (s *SmartContract) CcGetResult() (string, error)
6 func (s *SmartContract) CcVote(..  proofObj ZokProof, value string)
    (string, error)
7
8 }
```

En donde ”ccSetup” es la función que inicializa los artefactos y los guarda en cadena. ”CcAddId” Es la función que registra una nueva identidad al padrón. ”ccGetArtifacts” es la que retorna los artefactos necesarios para crear una prueba. ”ccGetResult” retorna el resultado parcial de la votación y ”ccVote” es la función que registra un voto. Para el caso de ”ccGetArtifacts” y ”ccGetResult” se utilizan simples queries a couchDB.

La función ”ccSetup” revisa que, tanto el padrón de votantes como, también, la llave de verificación, no hayan sido inicializados. Si nada ha sido inicializado, se procede a procesar estos datos y guardarlos en la cadena.

La función ”ccVote” es la más complicada. Primero, obtiene el padrón y la llave de verificación mediante ”queries” a la base de datos. Luego, procesa y construye las estructura de la prueba y las entradas públicas de la computación. Revisa que el padrón presente en la prueba corresponda a la inscrita en el proceso de inicialización y que el ”nullifier” no haya sido utilizado anteriormente. Por último, verifica la prueba mediante una función que implementa el algoritmo de verificación de Groth16 de la ecuación 6.4:

```
1    func verify(vk VerKey, proof Proof, inputs []*big.Int) bool {
2  vkx, errvk := makeVkX(inputs, vk)
3  if errvk != nil {
4    log.Printf("Error obteniendo Vk_x: %v", errvk)
5    return false
6  }
7  var g1P []*bn256.G1
8  var g2P []*bn256.G2
9  g1P = append(g1P, &proof.A, new(bn256.G1).Neg(vkx), new(bn256.G1).
    Neg(&proof.C), new(bn256.G1).Neg(&vk.Alpha))
```

```
  g2P = append(g2P, &proof.B, &vk.Gamma, &vk.Delta, &vk.Beta)

  return bn256.PairingCheck(g1P, g2P)
}

func makeVkX(inputs []*big.Int, vk VerKey) (*bn256.G1, error) {
  vkx := parseG1Point([]string{"00", "00"})
  if len(inputs)+1 != len(vk.Gabc) {
    return vkx, errors.New("Error en el tamano de los inputs")
  }
  for i, _ := range inputs {
    adder := new(bn256.G1).ScalarMult(&vk.Gabc[i+1], inputs[i])
    vkx = new(bn256.G1).Add(vkx, adder)
  }
  vkx = new(bn256.G1).Add(vkx, &vk.Gabc[0])
  return vkx, nil
}
```

Con el ”chaincode” listo se puede implementar el sistema de votación con la infraestructura de Hyperledger Fabric, presentada en el capítulo 4, y la generación de votos presentada en la sección anterior.

9. RESULTADOS

Una vez que el sistema fue construido, se le hicieron distintas pruebas para poder concluir si los objetivos propuestos fueron cumplidos. Las pruebas que se hicieron, se describen a continuación:

- Pruebas de funcionamiento: con estas pruebas se comprobó que el sistema cumple con el requerimiento mínimo de diseño.
- Pruebas de eficiencia de función Hash: en esta prueba se probaron las alternativas de funciones hash y su desempeño en el sistema, debido al rol crucial que cumplen en este.

9.1 Pruebas de funcionamiento

Con esta prueba se desea probar el funcionamiento básico del sistema, así como el que cumpla con las reglas establecidas en los objetivos. Para ello, esta prueba muestra el flujo del sistema mediante la interacción de las interfaces del este. Los dos componentes revisados son: la consola de votación y la base de datos de uno de los peers de la red de Hyperledger, ya que estas son las interfaces que permiten interactuar directamente con la cadena.

9.1.1 Resultados esperados

Se espera que el sistema permita iniciar la cadena y, con ello, que incluya toda la información necesaria, por lo cual, en la base de datos debe aparecer el padrón electoral y la llave de verificación. Posteriormente, la consola debe permitir generar un voto

al usar el secreto de algún votante inscrito. El sistema debe fallar si se intenta votar nuevamente con la mismo secreto o votar con un secreto inválido. La prueba debe ser guardada en la base de datos junto con la elección del votante, cuando el voto sea válido.

9.1.2 Resultados obtenidos

La figura 9.1 muestra la vista de la consola de administración que permite iniciar la cadena. Al iniciar la cadena, el bloque génesis y los primeros datos son creados y, con ellos, se inicia la base de datos de couchDB. El padrón electoral que contiene además la llave de verificación y la constante del sistema se muestra en la imagen 9.3.

La imagen 9.4 muestra la vista de la consola de votación. Al ingresar una elección y un secreto perteneciente a un votante inscrito se genera un voto. La imagen 9.5 evidencia como se agrega un voto en la base de datos. Si se intenta volver a votar con el mismo secreto un error es generado del lado de Hyperledger la cual es devuelta a la consola de votación como se aprecia en la imagen 9.6. Si por otro lado se utiliza una llave inválida aparece el error mostrado en la imagen 9.7 la cual es generada en la misma consola. El mensaje evidencia que matemáticamente la prueba ZK-SNARK falló en generarse al no cumplirse el código de votación descrito en la sección 8.2.

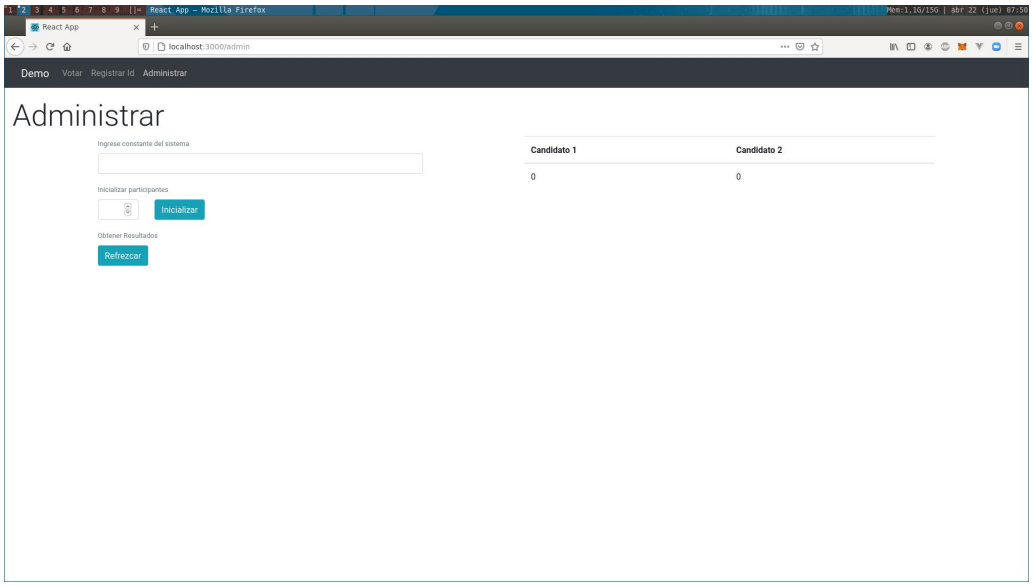


Fig. 9.1: Vista de consola de administración. Con esta vista se da la orden de iniciar la cadena. El bloque génesis es generado y, con ello, las primeras transacciones que incluyen los elementos iniciales necesarios para que el sistema funcione.

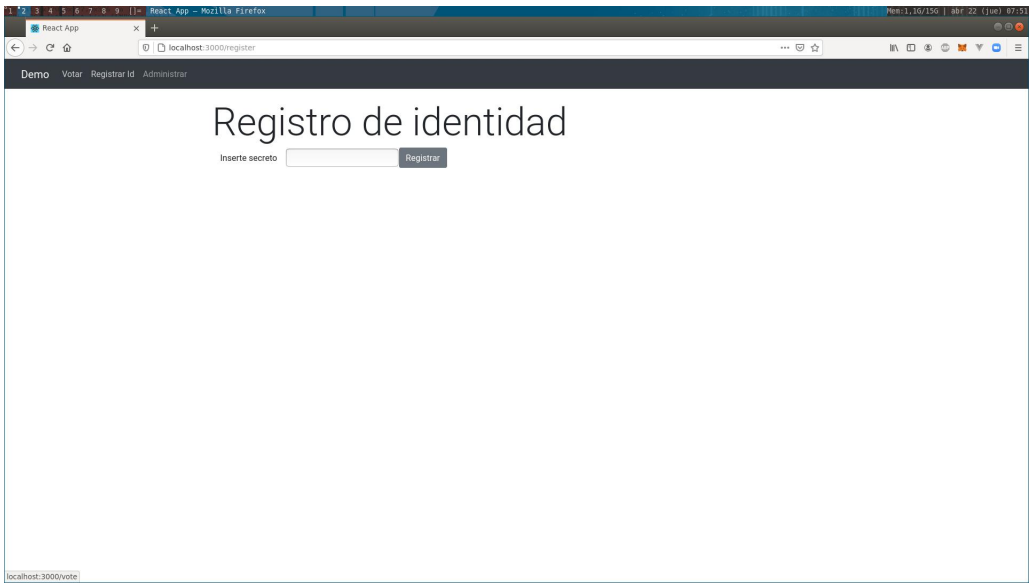


Fig. 9.2: Vista de consola de registro de id.

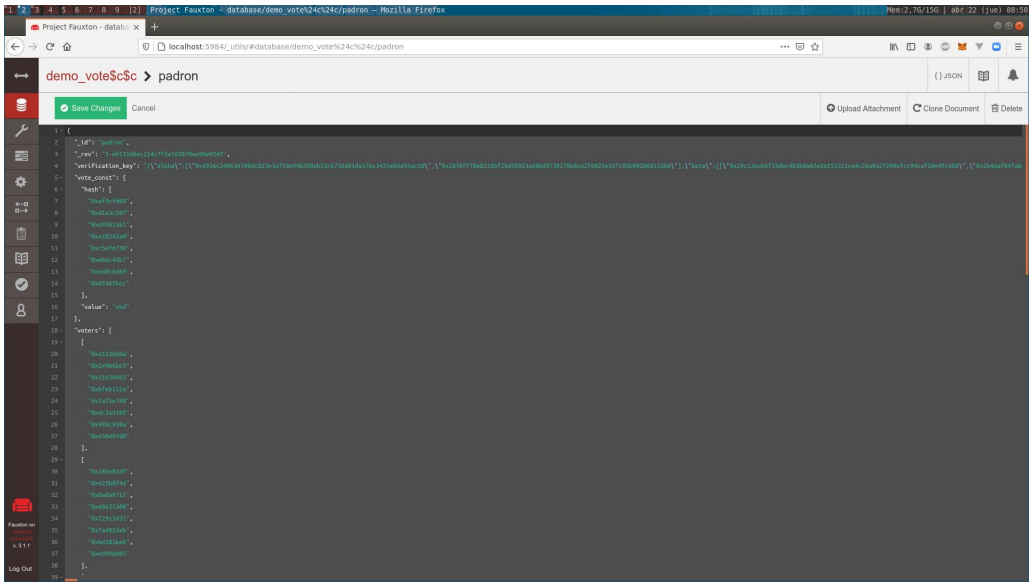


Fig. 9.3: Vista de couchDB de los datos iniciales cargados a la cadena.

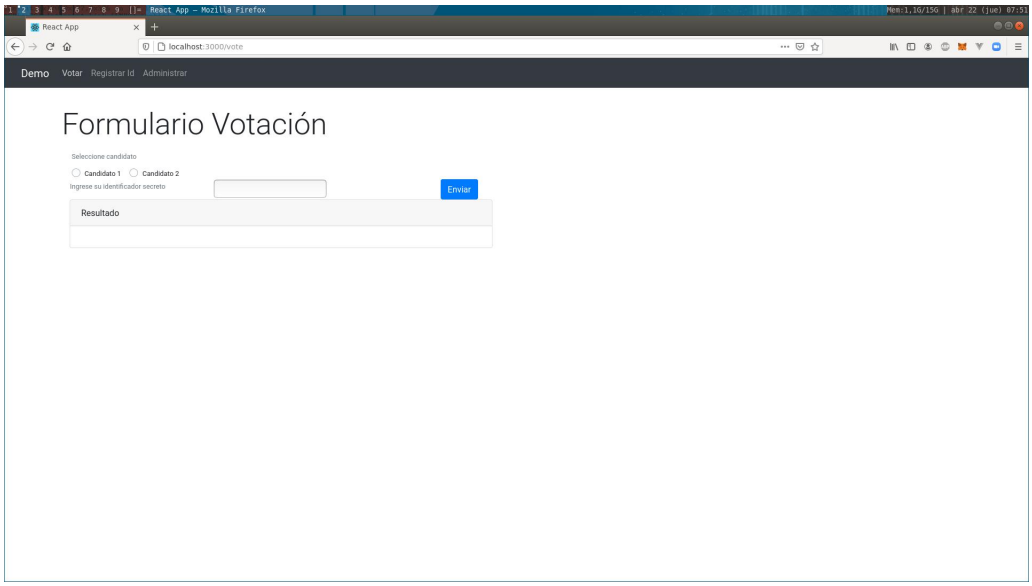


Fig. 9.4: Vista de consola de votación. Se muestra la elección de un candidato y el ingreso de una secreto de algún votante habilitado.

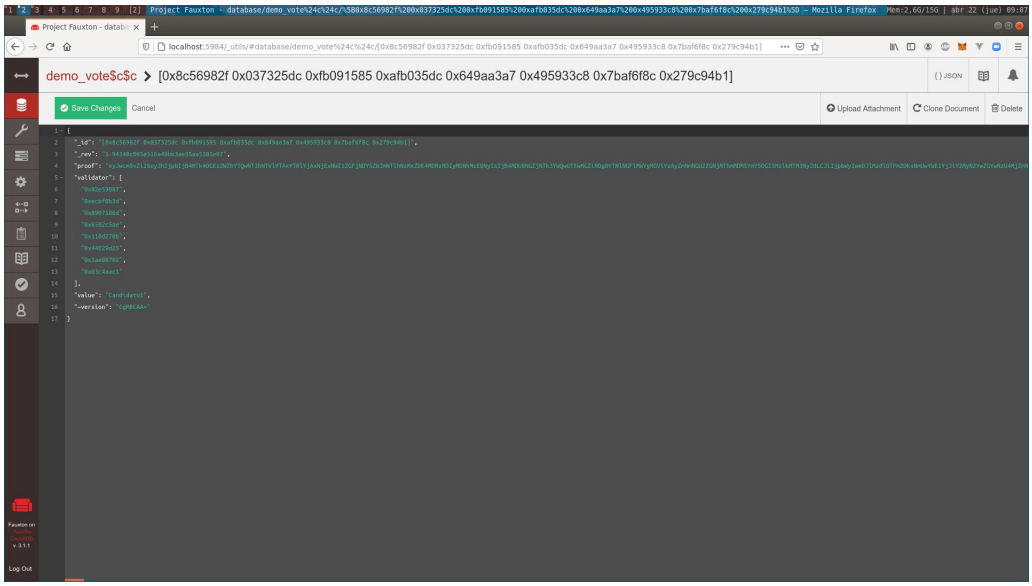


Fig. 9.5: Se aprecia que se agrega una entrada en la base de datos con el voto válido emitido.

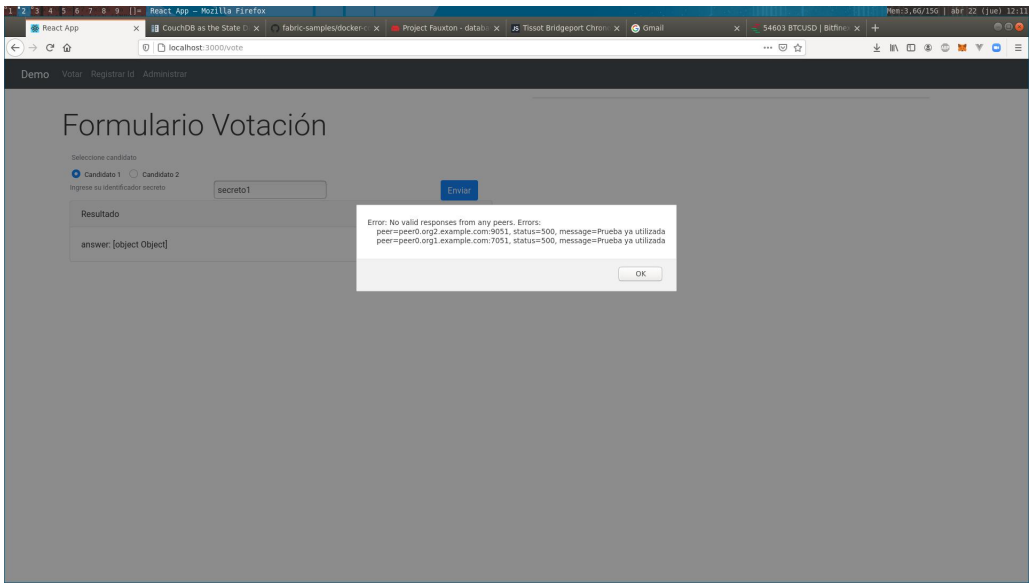


Fig. 9.6: Log de error al intentar votar nuevamente con el mismo secreto del votante.

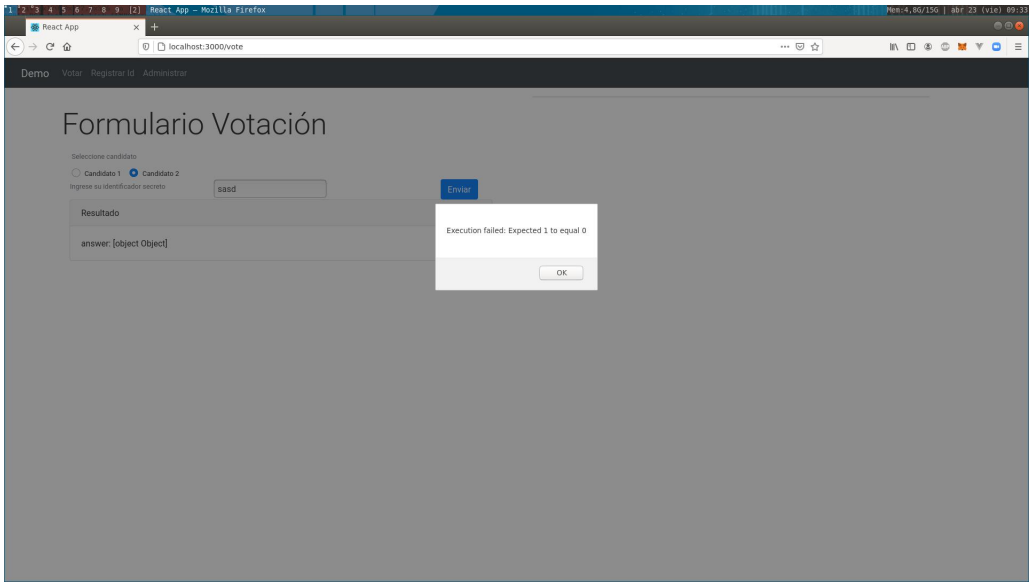


Fig. 9.7: Log de error al intentar usar un secreto no registrado.

9.2 Eficiencia de la función Hash

Zokrates ofrece dentro de sus librerías del DSL dos funciones Hash: SHA2 y la función Pedersen. SHA2 es uno de los protocolos más probados y comunes en cuanto a funciones Hash, debido a su conocida resistencia a ataques de colisiones como se estudia en [60]. Mientras que Pedersen es un protocolo que se comenzó a estudiar a propósito de la búsqueda de algoritmos eficientes para las ZK-SNARKs.

El presente trabajo utiliza extensivamente las funciones Hash en la implementación de la computación para la creación de pruebas de cero conocimiento, como se explica en la sección 8.2. Por lo tanto, la elección de una función Hash es de mucha importancia, en términos de la seguridad y eficiencia del sistema.

Un estudio de seguridad comprende un extenso trabajo que se escapa de los alcances de la presente memoria. Sin embargo, un estudio de la eficiencia es bastante directo, al sólo necesitar saber en cuántas compuertas aritméticas del QAP crea cada aplicación de función Hash.

Con el objetivo de determinar la eficiencia del sistema, se procedió a realizar prue-

bas con 4 rondas, en donde se comparó un código implementando la función SHA2 y otro implementando Pedersen, ambos con la estructura de la computación de la sección 8.2. En cada ronda se le agregan 2 funciones hash extras al código, representando a un crecimiento de 2 nodos del ”path” del árbol de Merkle del votante. En cada ronda se midió la cantidad de compuertas aritméticas generadas, el tamaño del QAP generado y el tamaño de la ”proving key” generada.

9.2.1 Resultados

Los resultados se pueden observar en las imágenes 9.8, 9.9 y 9.10. Estos datos muestran que en promedio las computaciones que emplean SHA2 genera 3.89 veces más compuertas, su QAP pesa 4.02 veces más y su ”proving key” pesa 3.47 veces más que las que implementan el Hash de Pedersen.

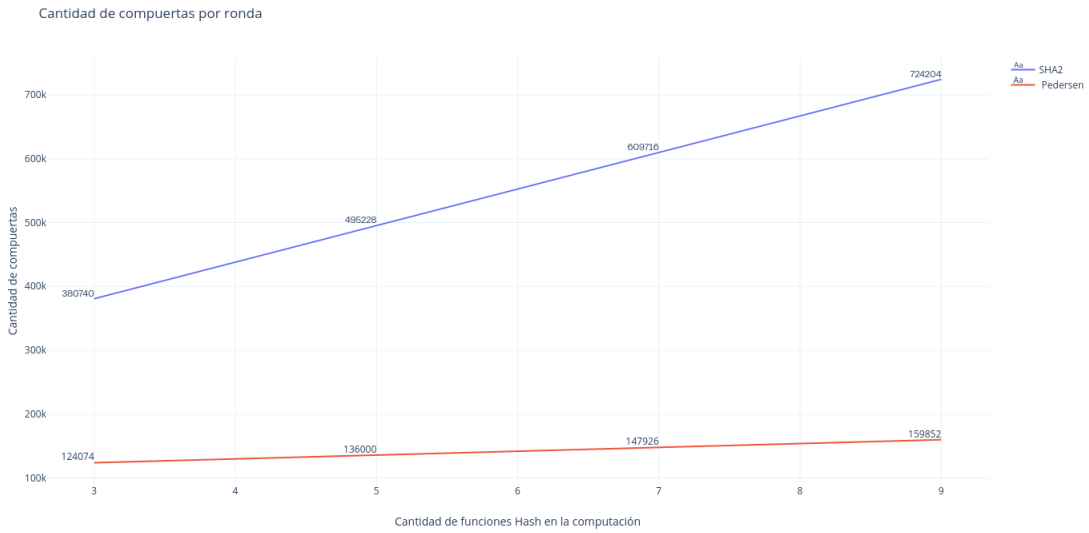


Fig. 9.8: Comparación de cantidad de compuertas generadas por las computaciones al tener implementadas 3,5,7 y 9 funciones Hash SHA2 y Pedersen.

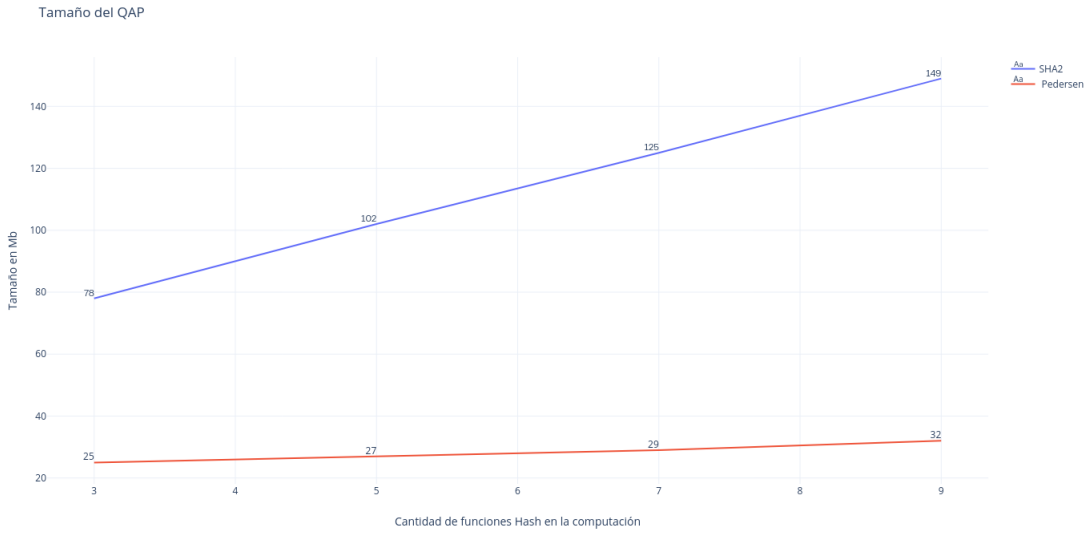


Fig. 9.9: Comparación del peso del QAP generadas por las computaciones que implementan 3,5,7 y 9 funciones Hash SHA2 y Pedersen.

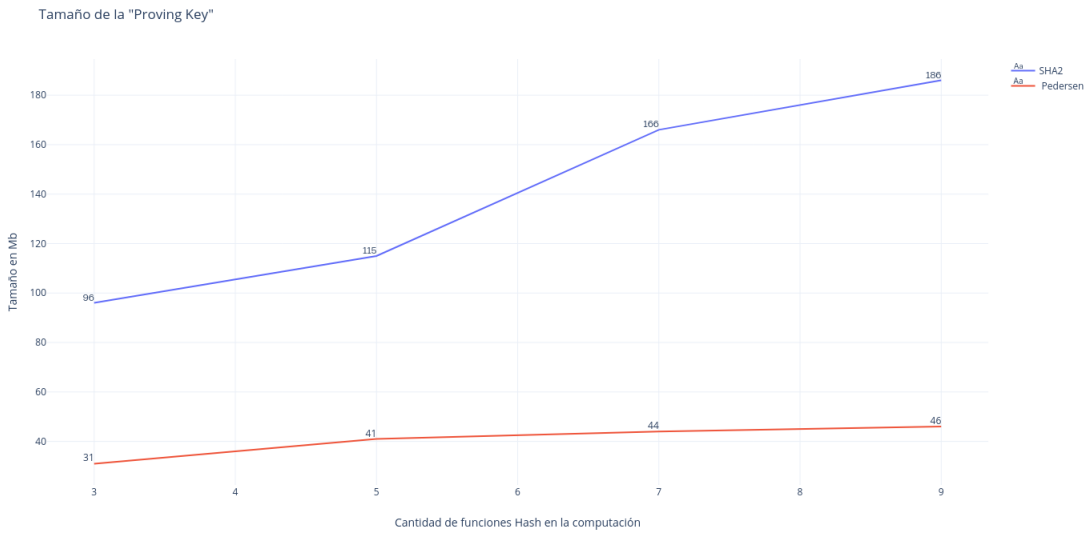


Fig. 9.10: Comparación del peso de las ”Proving Key” generadas por las computaciones que implementan 3,5,7 y 9 funciones Hash SHA2 y Pedersen.

10. CONCLUSIÓN

En la presente memoria se planteó un sistema integral que provee un servicio específico de votación electrónica. Se detalló cómo construirlo y finalmente se le realizaron distintas mediciones sobre funcionamiento y escalabilidad.

A continuación se presentará el balance entre los objetivos propuestos y los resultados logrados. Se discutirá sobre temas estrechamente relacionados, pero que se quedan afuera del alcance de la memoria. Finalmente se propondrán temas para la continuación del trabajo en el futuro.

10.1 Revisión de resultados y objetivos

10.1.1 Objetivos cumplidos

Los resultados de funcionamiento dan cuenta que el sistema cumple con la funcionalidad básica. Los votantes pueden sufragar sólo una vez. Las pruebas de cero conocimiento otorgan anonimato y verificación ”end to end”. Mientras que la publicación de estas en la cadena de bloques otorgan verificación pública e integridad de datos.

La escalabilidad de la infraestructura de Hyperledger no fue probada en esta memoria, pero existen extensos trabajos de benchmarking a Hyperledger Fabric, como [62]. En este trabajo se muestra que con una correcta política de ”endorsing” se pueden lograr un buen ratio de transacciones, que bordean las 100 a 200 transacciones por segundo en sus pruebas, independientes del tamaño de las organizaciones y su cantidad, lo que da a entender de que puede soportar grandes flujos de transacciones como ocurriría en una votación general.

10.1.2 Objetivos no satisfechos

En esta memoria no se pudo abordar el objetivo de ”Fairness”. Si bien, para su cumplimiento basta con la encriptación de los resultados y su desencriptación durante el conteo, la implementación en Hyperledger no es trivial por lo que se prefirió la implementación del resto de funcionalidades dejando fuera a esta.

Por otro lado, se desistió de implementar Forgivness/Resistencia a la coerción porque la solución no es sencilla al ocupar zk-SNARK. La única manera de acusar que un votante ya votó es mediante el ”nullifier”, el cual *debe ser único*. Al ser pruebas públicas, el atacante puede darse cuenta que su víctima cambió de elección al usar el mismo nullifier. Si el nullifier cambia, no hay forma segura de acusar que la elección anterior del votante fue reemplazada.

Una posible solución, a la posibilidad de coerción, es incluir en la lógica de la computación de la zk-SNARK una prueba de que la votación no ha sido reemplazada y que los votos que la reemplacen invaliden dicha prueba. Sin embargo, esta lógica es difícil de implementar sin afectar la seguridad de los votos.

10.1.3 Balance y comentarios finales del sistema

En general, los objetivos fueron cumplidos, logrando así implementar con éxito un sistema de votación electrónico basado en ”blockchain” y pruebas de conocimiento cero. Este sistema cumple con las características de un piloto para una prueba de concepto, pero su implementación en un contexto real debe vencer diversos desafíos.

Los desafíos se concentran en la estructura de la consola de votación, tema que fue dejado de lado en esta memoria al implementarse una, cuyo único propósito era la interacción con el sistema, pero que no cumple con lo necesario para una implementación segura y funcional en la realidad.

Otro punto a considerar, en una implementación real, es la decisión de la infraestructura de Hyperledger Fabric. Las reglas de verificación de las transacciones, la cantidad de organizaciones y sus tamaño, la elección de sus base de datos, la in-

fraestructura de red que las sostiene, etc.

Sin embargo, a pesar de estos desafíos este trabajo concluye presentando una alternativa viable de sistema de votación en un contexto en donde la discusión es cada vez más recurrente y polémica.

10.2 Discusión e Implementación en producción

En esta memoria se dejaron varios puntos sin discutir que están directamente vinculados con el tema, pero cuya participación es secundaria. Además, el objetivo principal de esta memoria era ofrecer un piloto experimental que funcionara de directriz para una implementación real. A continuación, se discutirán los temas que tienen incidencia en el sistema final y cuya decisión de diseño puede afectar a la eficiencia y funcionamiento del sistema real.

10.2.1 Consola de Votación

En el sistema se asumió la existencia de una consola de votación, la cual fue implementada en el mismo cliente de Hyperledger. En una implementación real esto *nunca debe suceder*. Es esencial que las credenciales del usuario nunca salgan de la consola por lo que una aplicación que transfiere las credenciales hacia un servidor externo no sirve al propósito del sistema.

Una posibilidad es tener consolas estáticas, en lugares de votación que se encarguen de calcular las pruebas y sólo liberen la prueba creada y la elección del votante.

Otra posibilidad es crear aplicaciones web o móviles que calculen las pruebas. Estos es más complejo porque generalmente estas aplicaciones deben ser escritas en lenguajes y para plataformas que no dan el mejor soporte a las herramientas necesarias para la creación de las ZK-SNARKs. Una posible solución es el uso de ZoKrates mediante WebAssembly en una aplicación web. ZoKrates (que está escrito en Rust) puede ser compilado a WebAssembly para que el Browser lo interprete nativamente sin la necesidad de contactar una API externa.

Esta opción ofrece una mayor movilidad y mejor manejo de la aplicación, pero su implementación tiene otros retos al estar parcialmente implementada en ”Rust” y ”WebAssembly”.

10.2.2 Arquitectura Hyperledger Y Federación de cadena

Un aspecto importante de la escalabilidad y que no fue discutido profundamente es la arquitectura óptima de Hyperledger Fabric para esta aplicación. Como se mostró en los resultados el benchmark hecho en [62] sugiere una cantidad máxima 150 tps para su implementación, es decir, el sistema se estabiliza en una cierta cantidad de tps. Además sugiere el uso de CouchDB sólo si se usan llaves compuesta, pero señala que el uso de levelDB aumenta la eficiencia del sistema. También señala que las reglas más flexibles y los tamaños de bloques más pequeños disminuyen la latencia cuando la cantidad de tps se satura.

Ante una eventual saturación del sistema por muy distribuido que esté conviene evaluar la federación de la cadena, es decir, crear distintos canales asignando a distintos nodos para que la recepción y el conteo de votos se haga de forma paralela. Esto debe ser evaluado dependiendo de la cantidad de árboles de Merkle creados y la cantidad de organizaciones y nodos haya disponible.

Por otra parte se debe tener en cuenta el tamaño mismo de los árboles de Merkle. Un árbol muy grande puede hacer que la llave de generación de pruebas pesen más y el tiempo de generación de estas aumente. Pero un árbol muy pequeño puede comprometer el anonimato de sus miembros al existir la posibilidad de que pocos miembros acudan a votar y se pueda sacar una estadística o mapa de probabilidad de cuál fue su elección.

10.2.3 Pruebas de cero conocimiento

Para esta memoria se trabajó con el esquema de pruebas de cero conocimiento ZK-SNARK, presentado en el capítulo 6. ZK-SNARK presenta las ventajas de ser el más

rápido de verificar y el que mayor soporte tiene al momento. Son estas características por las cuales se tomó le decisión de usarlo. Pero ZK-SNARK tiene desventajas. Por una parte incluye un proceso de ”trusted-setup” el cual puede ser aprovechado y atacado para generar pruebas falsas. Por otra parte el uso de una CRS genera archivos que, según la complejidad de la computación, pueden ser de elevado tamaño.

Además sus ventajas cada vez son disminuidas al existir otros proyectos con alta actividad y desarrollo. Las otras alternativas a ZK-SNARK son ZK-STARK y Bulletproof. Ambas prescinden de un proceso de ”trastud setup” y no requieren una CRS. Sin embargo sus verificaciones y transporte puede ser más ineficiente. La imagen 10.1 muestra una comparación entre estas alternativas.

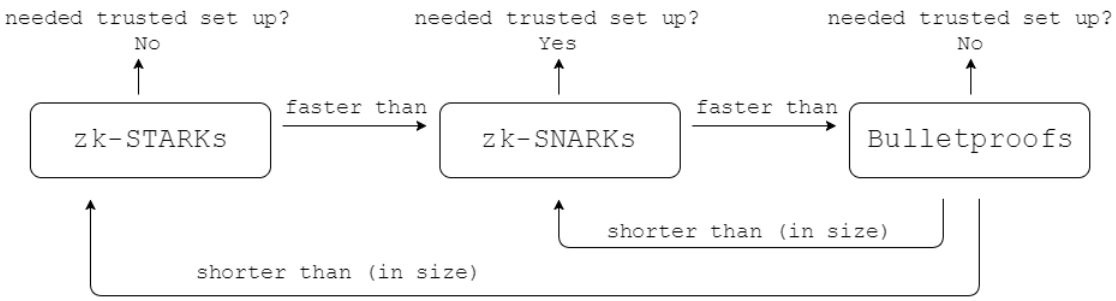


Fig. 10.1: Comparación entre ZK-SNARK, ZK-STARK y Bulletproof. Imagen proveniente de [63]

10.3 Trabajo Futuro

Debido a la extensión de este trabajo hay varios aspecto que pueden explorarse. Algunos de ellos dan a lugar nuevos trabajos que pueden presentar una mejoría o innovar en el presente trabajo. A continuación se explora alguno de estos temas.

10.3.1 Fairness y Resistencia a Coerción

Este trabajo no cumplió con dos de las características del sistema que se plantearon en los objetivos. Es necesario explorar alternativas de solución para lograr Fairness y resistencia a la Coerción.

Posibles soluciones a estos temas fueron presentados en la sección 10.1.2. Se necesita explorar un sistema de encriptación para los datos guardados en la cadena antes de la votación en donde la llave de desencriptación sea compuesta de tal forma que varios miembros de un conglomerado deban consensuar el momento de liberar la data para el conteo. Para esto se pueden ocupar ”Dynamic Threshold Public Key Encription” presentado en [64].

Por otra parte se debe encontrar una forma de que los votantes puedan anular y reemplazar su voto para evitar la coerción. Una forma es incluir en la computación una prueba de que el voto pertenece a una tabla que señala su unicidad. Luego, si un votante quiere deprecar su voto publicará otro que actualizará dicha tabla invalidando la prueba de su voto anterior, así durante el conteo de voto dicha prueba será invalidada y el voto no será contado. Sin embargo se debe tener cuidado con una implementación de un mecanismo así ya que puede generar una puerta trasera y amenazar el anonimato de los votantes o puede invalidar votos genuinamente emitidos.

10.3.2 ZK-STARK

Como se discutió en la sección 10.2.3 se puede investigar otros esquemas de pruebas de cero conocimiento. Una alternativa inmediata y que cada vez se facilita su implementación es las llamadas ZK-STARK (Zero Knowledge - Scalable Transparent Arguments of Knowledge) con características parecidas a las SNARks pero que su planteamiento matemático (cuyo enfoque difiere drásticamente con las SNARKs) permite crear pruebas sin la necesidad de un ”trusted setup”. Además se basan en menos ”supuestos” criptográficos y son resistentes a la computación cuántica.

Puede ser interesante crear un sistema similar al planteado en esta memoria pero cuyas pruebas de cero conocimiento sean implementadas con el protocolo de ZK-STARK.

Existen distintas herramientas para ello, entre ellas se encuentran StarkWare y AirAssembly/AirScript/genStark [65, 66]. Estas herramientas permiten escribir computaciones, compilarlas y generar pruebas con el protocolo de ZK-STARK.

10.3.3 Quadratic Voting

En las sociedades existen dos mecanismo en los cuales las personas ejercen decisiones colectivas. En la primera ante varias alternativas de decisiones las personas realizan un voto donde manifiestan su intención de apoyar una en específico. La decisión que consigue una mayoría gana, por lo cual es necesario que para hacerlo ”justo” las personas tengan derecho a un solo voto. Por otro lado en la economía las personas eligen qué productos y servicios son los exitosos y necesarios al pagar más por ellos. En este caso el poder de decisión de las personas se basa en su riqueza individual. Aquellos que acumulan más riquezas tienen mayor poder de influencia en las decisiones de mercado.

Ambas formas tienen falencias. Por un lado en el sistema de una persona un voto, aquellos que se ven menos afectados o están menos interesados por el resultado tienen el mismo poder de decisión que aquellos a los cuales la decisión los afecta directamente o están más interesados en el resultado. Por otro lado en el sistema basado en riquezas personales si bien las personas deben sacrificar una mayor riqueza para que sus intereses sean conseguidos aquellos que acumulan riquezas tienen una ventaja sobre aquellos que no.

El concepto de ”Quadratic Voting” es una propuesta que media entre ambos problemas, entre el deseo de los individuos de proteger sus propios intereses y la constante pugna de las sociedades por resolver el problema de los comunes. En el sistema de ”Quadratic Voting” las personas pueden votar cuantas veces quieran, como en el sistema económico, con la diferencia que cada voto extra tiene un costo cuadráticamente mayor, es decir, costará 2 veces más que el voto anterior. Las implicancias de esto, además de una explicación más profunda, se pueden leer en [67] y una explicación más técnicamente amistosa se puede leer la página de Vitalik Buterin [68].

Un sistema de ”Quadratic Voting” puede ser beneficiado por el sistema propuesto en esta memoria al necesitar transparencia, seguridad y anonimato, tal cual un sistema de voto convencional. Sólo se necesita programar ”créditos” de los votantes que

consumirán de forma cuadrática cada vez que voten. Se debe programar, además, alguna lógica de verificación que los créditos consumidos en un voto le pertenece al votante que lo emitió . Para esto se necesita un mecanismo similar al que es necesario para solucionar el problema de coerción, tan solo que en vez de que los nuevos votos reemplacen a los anteriores estos se sumen entre ellos.

BIBLIOGRAPHY

[1] SATOSHI Nakamoto, Bitcoin: A Peer-to-Peer Electronic Cash System, [en línea] Cryptography Mailing list at metzdowd.com, mayo 2009, <https://bitcoin.org/bitcoin.pdf > , [consulta: 25 octubre 2019].

[2] LAMPORT Leslie, Shostak Robert, Pease Marshall. The Byzantine generals problem. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 4, no 3, p. 382-401, julio 1982.

[3] GEORGIOS Konstantopoulos,Understanding Blockchain Fundamentals, Part 1: Byzantine Fault Tolerance,[en línea], medium.com, <https://medium.com/loom-network/understanding-blockchain-fundamentals-part-1-byzantine-fault-tolerance-245f46fe8419 >, [Consulta: 10 noviembre 2019].

[4] CASTRO Miguel, et al. Practical Byzantine fault tolerance. OSDI vol 99, No 1999 p. 173-186, febrero 1999.

[5] GAVIN Andresen, Bip 16: Pay to Script Hash, [en línea], Bitcoin's Github project BIPs, marzo 2012, <https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki>, [consulta: 10 de noviembre 2019]

[6] SCRIPT, [en línea], Bitcoin Wiki, julio 2019, <https://en.bitcoin.it/wiki/Script>, [consulta: 10 noviembre 2019]

[7] CONTRACT, [en línea], Bitcoin Wiki, febrero 2019, <https://en.bitcoin.it/wiki/Contract>, [consulta: 11 noviembre 2019]

[8] BUTERIN Vitalik, A Prehistory of the Ethereum Protocol [en línea], Vitalik's personal page, Septiembre 2017, <https://vitalik.ca/general/2017/09/14/prehistory.html>,[Consulta: 15 de noviembre 2019].

[9] WOOD Gavin, ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER BYZANTIUM VERSION 7e819ec - 2019-10-20, [en línea], Ethereum Foundation, Octubre 2019, <https://ethereum.github.io/yellowpaper/paper.pdf >, [Consulta: 17 noviembre]

[10] BUTERIN Vitalik, et al. "A next-generation smart contract and decentralized application platform". white paper, 2014, vol. 3, pag 37.

[11] ETHSTAT, [en línea], <https://ethstats.net/>, [Consulta: 17 noviembre 2019]

[12] ETHEREUM, Solidity v0.5.13 [en línea], <https://solidity.readthedocs.io/en/v0.5.12/ >, [Consulta: 17 noviembre 2019]

[13] BUTERIN Vitalik, Ethereum: The Ultimate Smart Contract and Autonomous Corporation Platform on the Blockchain, [en línea], Vitalik's personal page, Diciembre 2013, < https://web.archive.org/web/20131219030753/http://vitalik.ca/ethereum.html >, [Consulta: 17 Noviembre 2019]

[14] VUKOLIĆ Marko, Rethinking permissioned blockchains, En su: Proceedings of the ACM Workshop on Blockchain, Zurich, ACM, 2017. p. 3-7.

[15] QUORUM, Quorum Whitepaper, [en línea], Repositorio Github de JPMorgan Chase, Agosto 2018, < https://github.com/jpmorganchase/quorum/blob/master/docs/Quorum Whitepaper v0.2.pdf >, [Consulta: 18 noviembre 2019]

[16] HYPERLEDGER, An Introduction to Hyperledger, [en línea],hyperledger.org, Agosto 2018, < https://www.hyperledger.org/wp-content/uploads/2018/08/HL_Whitepaper_IntroductiontoHyperledger.pdf >, [Consulta: 19 noviembre 2019]

[17] Docker, Página web de docker, [en línea], Docker webpage, noviembre 2020, <https://https://www.docker.com/>, [consulta: 10 de noviembre 2020]

[18] ANDROULAKI Elli, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. En su: Proceedings of the Thirteenth EuroSys Conference at Porto Portugal, New York USA, ACM, abril 2018. p. 30.

[19] HYPERLEDGER Fabric, Chaincode for Operators, [en línea], hyperledger fabric readthedocs, < https://hyperledger-fabric.readthedocs.io/en/release-1.4/chaincode4noah.html >,[Consulta: 19 noviembre 2019]

[20] HYPERLEDGER Fabric, Glosary, [en línea], hyperledger fabric fabrictestdocs, < https://fabrictestdocs.readthedocs.io/en/latest/glossary.html >, [Consulta: 19 noviembre 2019]

[21] HYPERLEDGER Fabric, Membership Service Providers (MSP), [en línea], hyperledger fabric readthedocs, < https://hyperledger-fabric.readthedocs.io/en/release-1.4/msp.html >, [Consulta: 19 noviembre 2019]

[22] HYPERLEDGER Fabric, Policies in Hyperledger Fabric, [en línea], hyperledger fabric readthedocs, < https://hyperledger-fabric.readthedocs.io/en/release-1.4/policies.html >, [Consulta: 19 noviembre 2019]

[23] HYPERLEDGER Fabric, CouchDB as the State Database, [en línea], hyperledger fabric readthedocs, < https://hyperledger-fabric.readthedocs.io/en/release-1.4/couchdb_as_state_database.html >, [Consulta: 19 noviembre 2019]

[24] HYPERLEDGER Fabric, Building Your First Network, [en línea], hyperledger fabric readthedocs, < https://hyperledger-fabric.readthedocs.io/en/release-1.4/build_network.html >, [Consulta: 23 noviembre 2019]

[25] HYPERLEDGER Fabric, Prerequisites, [en línea], hyperledger fabric readthedocs, < https://hyperledger-fabric.readthedocs.io/en/release-1.4/prereqs.htmll >, [Consulta: 23 noviembre 2019]

[26] hyperledger / fabric-peer : 1.4.2, [en línea], Docker Hub, < https://hub.docker.com/layers/hyperledger/fabric-peer/1.4.2/images/sha256-c73650b28f3ac5860dc4121a3a5ea5d53ca2d6da6c05d5592d9993465cbab714 >, [Consulta: 22 noviembre 2019]

[27] GOLDWASSER Shafi, Silvio Micali, and Charles Rackoff. "The knowledge complexity of interactive proof systems." SIAM Journal on computing, 1981, vol 18, no 1 pag: 186-208.

[28] GOLDREICH Oded; Oren Yair. "Definitions and properties of zero-knowledge proof systems". Journal of Cryptology, 1994, vol. 7, no 1, pag: 1-32.

[29] DE SANTIS Alfredo, Micali Silvio; Persiano Giuseppe. "Non-interactive zero-knowledge proof systems". En "Conference on the Theory and Application of Cryptographic Techniques". Springer, Berlin, Heidelberg, 1987. pag: 52-72.

[30] BEN-SASSON Eli, et al. "Succinct non-interactive zero knowledge for a von Neumann architecture". En 23rd USENIX Security Symposium (USENIX Security 14). 2014. pag 781-796.

[31] BELLARE Mihir, PALACIO Adriana. "The knowledge-of-exponent assumptions and 3-round zero-knowledge protocols". En Annual International Cryptology Conference. Springer, Berlin, Heidelberg, 2004. pag. 273-289.

[32] GENNARO Rosario, et al. ”Quadratic span programs and succinct NIZKs without PCPs”. En ”Annual International Conference on the Theory and Applications of Cryptographic Techniques”. Springer, Berlin, Heidelberg, 2013. pag 626-645.

[33] SRINIVASA Rao, Subramanya Rao, ”Elliptic Curve Arithmetic for Cryptography” Tesis (Doctorado en Filosofía) Australia, The Australian National University, Agosto 2017, p.158.

[34] BLAKE, Ian, et al. ”Elliptic curves in cryptography”. Cambridge university press, 1999.

[35] PARNO Bryan, et al. ”Pinocchio: Nearly practical verifiable computation”. En 2013 IEEE Symposium on Security and Privacy. IEEE, 2013. p. 238-252.

[36] MAYER Hartwig. ”zk-SNARK explained: Basic Principles”. [en línea], blog.coinfabrik.com, 2016. < https://blog.coinfabrik.com/wp-content/uploads/2017/03/zkSNARK-explained_basic_principles.pdf >, [Consulta: 5 Diciembre 2019]

[37] BUTERIN Vitalik, ”Quadratic Arithmetic Programs: from Zero to Hero”,[En línea],medium blog Diciembre 2016, <https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649> [Consulta: 7 Diciembre 2019]

[38] BUTERIN Vitalik, ”Zk-SNARKs: Under the Hood”,[En línea],medium blog Febrero 2017, <https://medium.com/@VitalikButerin/zk-snarks-under-the-hood-b33151a013f6> [Consulta: 7 Diciembre 2019]

[39] BUTERIN Vitalik, ”Exploring Elliptic Curve Pairings”,[En línea],medium blog Enero 2017, <https://medium.com/@VitalikButerin/exploring-elliptic-curve-pairings-c73c1864e627> [Consulta: 7 Diciembre 2019]

[40] GROTH Jens, ”On the size of pairing-based non-interactive arguments”. En Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, Berlin, Heidelberg, 2016. pag. 305-326.

[41] JIANG Xin, ”How to Generate a Groth16 Proof for Forgery”, [En línea], medium blog, Noviembre 2019, <https://medium.com/ppio/how-to-generate-a-groth16-proof-for-forgery-9f857b0dcafd> [Consulta: 3 de Enero 2020]

[42] ZCash, ”Parameter Generation,[En línea],ZCash site,2018, <https://z.cash/technology/paramgen/> [Consulta: 8 Diciembre 2019]

[43] MERKLE Ralph C. ”A digital signature based on a conventional encryption function”. En Conference on the theory and application of cryptographic techniques. Springer, Berlin, Heidelberg, 1987. pag. 369-378.

[44] SZYDLO Michael. ”Merkle tree traversal in log space and time”. En International Conference on the Theory and Applications of Cryptographic Techniques. Springer, Berlin, Heidelberg, 2004. pag 541-554.

[45] HOPWOOD Daira, et al. ”Zcash protocol specification”. Tech. rep. 2016–1.10. Zerocoin Electric Coin Company, Tech. Rep., 2016.

[46] SASSON Eli Ben, et al. Zerocash: ”Decentralized anonymous payments from bitcoin”. En 2014 IEEE Symposium on Security and Privacy. IEEE, 2014. pag. 459-474.

[47] MAATEN Epp. ”Towards remote e-voting: Estonian case”. Electronic Voting in Europe-Technology, Law, Politics and Society, 2004, vol. 47, pag. 83-100.

[48] GIBSON J. Paul, et al. ”A review of e-voting: the past, present and future”. Annals of Telecommunications, 2016, vol. 71, no 7-8, pag 279-286.

[49] TARASOV Pavel, TEWARI Hitesh. ”Internet Voting Using Zcash”. IACR Cryptology ePrint Archive, 2017, vol. 2017, p. 585.

[50] HARDWICK Freya Sheer, et al. ”E-Voting with blockchain: an E-Voting protocol with decentralisation and voter privacy”. En 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). IEEE, 2018. pag 1561-1567.

[51] GAWEŁ Dawid, et al. ”Apollo–End-to-End Verifiable Internet Voting with Recovery from Vote Manipulation”. En International Joint Conference on Electronic Voting. Springer, Cham, 2016. p. 125-143.

[52] SPRINGALL Drew, et al. ”Security analysis of the Estonian internet voting system”. En Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM, 2014. p. 703-715.

[53] EBERHARDT Jacob, Tai Stefan. ”ZoKrates-Scalable Privacy-Preserving Off-Chain Computations”. En 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). IEEE, 2018. pag. 1084-1091.

[54] ZOKRATES, ”Tutorial: proving knowledge of a hash preimage”,[En línea],ZoKrates github,2018, <https://zokrates.github.io/sha256example.html#tutorial-proving-knowledge-of-a-hash-preimage> [Consulta: 18 Diciembre 2019]

[55] BERNSTEIN Daniel J., et al. ”High-speed high-security signatures”. Journal of Cryptographic Engineering, 2012, vol. 2, no 2, pag. 77-89.

[56] BERNSTEIN Daniel J., et al. Twisted edwards curves. En International Conference on Cryptology in Africa. Springer, Berlin, Heidelberg, 2008. pag. 389-405.

[57] WHITEHAT Barry WhiteHat, Baylina Jordi, Bell Marta, Baby Jubjub Elliptic Curve, Ethereum foundation, 2018.

[58] ZCash, ”What is JubJub?”, ZCash blog, 2018, <https://z.cash/technology/jubjub/> [Consulta: 26 Diciembre 2019]

[59] MEHTA Dinesh P., SAHNI Sartaj.”Handbook of data structures and applications”. Chapman and Hall/CRC, 2004.

[60] SANADHYA Somitra Kumar, SARKAR, Palash. ”A combinatorial analysis of recent attacks on step reduced SHA-2 family”. Cryptography and Communications, 2009, vol. 1, no 2, p. 135-173.

[61] BAYLINA Jordi, BELLÉS Marta. 4-bit Window Pedersen Hash On The Baby Jubjub Elliptic Curve.

[62] THAKKAR Parth, NATHAN Senthil, VISWANATHAN Balaji. ”Performance benchmarking and optimizing hyperledger fabric blockchain platform”. En 2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS). IEEE, 2018. p. 264-276.

[63] StackExchange, ”zk-SNARKs vs. Zk-STARKs vs. BulletProofs? (Updated)”,[En línea],ZoKrates ethereum stackexchange,2019, <https://ethereum.stackexchange.com/questions/59145/zk-snarks-vs-zk-starks-vs-bulletproofs-updated> [Consulta: 10 Marzo 2020]

[64] DELERABLÉE Cécile, POINTCHEVAL David. ”Dynamic threshold public-key encryption”. En Annual International Cryptology Conference. Springer, Berlin, Heidelberg, 2008. p. 317-334.

[65] GuildOfWeavers , ”AirAssembly”,[En línea],AirAssembly github,2020, <https://github.com/GuildOfWeavers/AirAssembly/tree/master/specs> [Consulta: 15 Marzo 2020]

[66] GuildOfWeavers , ”genSTARK”,[En línea],genSTARK github,2020, <https://github.com/GuildOfWeavers/genSTARK> [Consulta: 15 Marzo 2020]

[67] LALLEY Steven P., WEYL, E. Glen. ”Quadratic voting: How mechanism design can radicalize democracy”. En AEA Papers and Proceedings. 2018. p. 33-37.

[68] Vitalik Buterin , ”Quadratic Payments: A Primer”,[En línea],Vitalik Buterin’s Website,2020, <https://vitalik.ca/general/2019/12/07/quadratic.html> [Consulta: 15 Marzo 2020]