Universidad Zaragoza

Trabajo Fin de Grado

Aplicación de pruebas de conocimiento cero en la autenticación mediante *sum-check*

Application of Zero-Knowledge Proofs in Authentication Using Sum-Check

Autor

Jorge Ruiz González

Director

Ricardo Julio Rodríguez Fernández

Titulación del autor

Grado en Ingeniería Informática

Departamento

Departamento de Informática e Ingeniería de Sistemas

Escuela de Ingeniería y Arquitectura

Septiembre de 2026

Resumen

El objetivo principal de este trabajo consiste en el desarrollo de un prototipo de autenticación inspirado en técnicas utilizadas en el ámbito de conocimiento cero. Para ello, se diseña e implementa un prototipo cliente-servidor en el que la contraseña de los usuarios se transforma en una representación algebraica sobre un cuerpo finito. Sobre dicha representación se aplica una versión adaptada del protocolo *sum-check*, que permite verificar una afirmación algebraica derivada del secreto sin enviar directamente la contraseña al servidor.

El sistema desarrollado incluye una modalidad interactiva del protocolo y una modalidad no interactiva obtenida mediante la transformación de Fiat-Shamir. Además, se incorpora una extensión didáctica basada en el concepto de compromiso polinomial, con el objetivo de ilustrar algunos mecanismos presentes en construcciones zk-SNARK modernas.

La implementación se organiza como una aplicación web con cliente en navegador, servidor Node.js y persistencia en PostgreSQL. Se proporciona adicionalmente un despliegue mediante Docker para facilitar la reproducción del prototipo independientemente del equipo anfitrión. La evaluación incluye una validación funcional de los principales casos de aceptación y rechazo, así como una comparación de rendimiento entre las distintas configuraciones implementadas.

Los resultados muestran que la modalidad Fiat-Shamir reduce estructuralmente el número de peticiones frente al modo interactivo. También obtiene menores latencias en Safari y en Firefox sobre los dos equipos evaluados. En Edge y Chrome, ambos basados en Chromium, se observa en cambio un incremento del coste de la modalidad no interactiva a partir de determinados tamaños. La ejecución de Firefox 152.0.6 en macOS y Windows conserva el mismo orden relativo entre configuraciones, aunque con menores tiempos absolutos en el equipo macOS. El caso de 256 bytes se interpreta como una prueba de estrés y presenta una variabilidad especialmente elevada en Safari.

Finalmente, se analizan las limitaciones del sistema. El prototipo no proporciona conocimiento cero formal ni seguridad criptográfica equivalente a la de un zk-SNARK real, y los datos de verificación almacenados no impiden el ensayo fuera de línea de contraseñas candidatas si se compromete la base de datos. Se ofrece, sin embargo, una aproximación académica y práctica a varias ideas fundamentales de estos sistemas, tales como aritmetización, protocolos algebraicos, eliminación de interacción, compromisos polinomiales y configuración inicial.

Abstract

The main objective of this work is to develop an authentication prototype based on zero-knowledge techniques. To this end, a client-server prototype is designed and implemented in which users' passwords are transformed into an algebraic representation over a finite field. An adapted version of the *sum-check* protocol is then applied to this representation, allowing the system to verify an algebraic statement derived from the secret without sending the password directly to the server.

The developed system includes an interactive mode of the protocol and a non-interactive mode obtained through the Fiat-Shamir transformation. In addition, a didactic extension based on the concept of polynomial commitment is incorporated, with the aim of illustrating some mechanisms present in modern zk-SNARK constructions.

The implementation is organized as a web application with a browser-based client, a Node.js server and PostgreSQL persistence. A Docker deployment is also provided to facilitate the reproduction of the prototype independently of the host machine. The evaluation includes a functional validation of the main acceptance and rejection cases, as well as a performance comparison between the different implemented configurations.

The results show that the Fiat-Shamir mode structurally reduces the number of requests compared to the interactive mode. It also achieves lower latencies in Safari and in Firefox on both evaluated hosts. In Edge and Chrome, which are both Chromium-based, the non-interactive mode instead shows a higher cost from certain input sizes onwards. Running Firefox 152.0.6 on macOS and Windows preserves the same relative ordering between configurations, although the macOS host obtains lower absolute times. The 256-byte case is treated as a stress test and shows particularly high variability in Safari.

Finally, the limitations of the system are analysed. The prototype does not provide formal zero-knowledge guarantees or cryptographic security equivalent to a real zk-SNARK, and the stored verification data does not prevent offline testing of password candidates if the database is compromised. However, it offers an academic and practical approach to several fundamental ideas of these systems, such as arithmetization, algebraic protocols, non-interactivity, polynomial commitments and trusted setup.

Declaración sobre el uso de inteligencia artificial

Durante la elaboración de este trabajo se han utilizado herramientas de inteligencia artificial generativa de OpenAI, principalmente ChatGPT (GPT-5.5 Thinking) y Codex, como apoyo para el análisis y la edición de fragmentos de código y diagramas. En las fases finales de revisión también se empleó GPT-5.6 Sol.

Las propuestas generadas por estas herramientas han sido revisadas, contrastadas y adaptadas por el autor. El diseño y la implementación del prototipo, la ejecución de las pruebas, la selección e interpretación de los resultados y la versión final del trabajo son responsabilidad del autor. Las herramientas de inteligencia artificial no se han utilizado como fuente. Las afirmaciones técnicas se apoyan en las referencias citadas en la memoria.

Índice general

Capítulo 1

Introducción

1.1. Motivación

La autenticación de usuarios constituye uno de los mecanismos fundamentales en la seguridad de los sistemas informáticos. En la mayoría de los sistemas actuales basados en una arquitectura cliente-servidor, el acceso a recursos protegidos se realiza mediante el envío de credenciales al servidor. Habitualmente, estas credenciales consisten en un identificador de usuario y una contraseña asociada.

En un diseño seguro, el servidor no debería almacenar las contraseñas de los usuarios, sino valores derivados de las mismas. Estos valores se pueden obtener mediante técnicas criptográficas, como funciones *hash* con *salt* o funciones de derivación de claves. Sin embargo, incluso en este escenario, la existencia de una base de datos que relaciona usuarios con valores derivados de sus contraseñas sigue constituyendo un objetivo atractivo para un atacante. Si dicha base de datos se ve comprometida, pueden realizarse ataques fuera de línea de diccionario o fuerza bruta, probando contraseñas candidatas hasta encontrar alguna que produzca el mismo valor almacenado [13].

Este problema motiva el estudio de mecanismos en los que un usuario pueda demostrar conocimiento de un secreto sin necesidad de revelarlo directamente. En este contexto aparecen las pruebas de conocimiento cero, introducidas formalmente por Goldwasser, Micali y Rackoff, que permiten a los usuarios demostrar la validez de una afirmación sin revelar información adicional más allá de dicha validez [1].

Se estudian algunos de los principales protocolos algebraicos relacionados con las pruebas de conocimiento cero. En particular, se verá cómo una contraseña puede representarse mediante una tabla de evaluaciones sobre un cuerpo finito y cómo una afirmación sobre dicha tabla puede verificarse mediante el protocolo *sum-check*, uno de los bloques fundamentales en el estudio de pruebas interactivas de conocimiento cero [4, 10].

A partir de esta idea inicial, el trabajo desarrolla varias capas incrementales. En primer lugar, se implementa una versión adaptada del protocolo *sum-check*. Posteriormente, se obtiene una versión no interactiva mediante la transformación de Fiat-Shamir, técnica que permite reemplazar los retos del verificador por valores derivados del registro de interacción del protocolo, mediante una función *hash* [2]. Finalmente, se incorpora una extensión basada en compromisos polinomiales. Los compromisos polinomiales permiten enlazar cada usuario a un polinomio y posteriormente demostrar evaluaciones concretas del mismo de forma eficiente sin revelarlo explícitamente [5].

Estas piezas permiten relacionar el prototipo desarrollado con la estructura general de los zk-SNARKs modernos, que combinan aritmetización, protocolos polinomiales, compromisos criptográficos y técnicas de no interacción [11]. Sin embargo, el prototipo resultante debe entenderse como una implementación académica y didáctica, no como una prueba de conocimiento cero formal ni como una solución criptográfica lista para producción. El autor de este trabajo no se hace responsable de los posibles compromisos de confidencialidad que puedan derivarse.

1.2. Objetivos

El objetivo principal de este trabajo es el diseño e implementación de un prototipo académico de autenticación inspirado en técnicas de conocimiento cero, destacando en particular el algoritmo *sum-check*. Se pretende verificar afirmaciones algebraicas derivadas de la representación de una contraseña, tomando como referencia el papel de este protocolo dentro de los sistemas de pruebas interactivas [10].

A partir de este objetivo general, se establecen varios objetivos específicos. En primer lugar, se busca estudiar los fundamentos de las pruebas de conocimiento cero y de las pruebas interactivas, así como analizar el protocolo *sum-check* y su importancia dentro de construcciones criptográficas modernas. También se plantea diseñar una aritmetización sencilla de una contraseña mediante su representación binaria y una tabla de evaluaciones sobre un cuerpo finito.

Desde el punto de vista de implementación, se pretende desarrollar una adaptación interactiva del protocolo *sum-check* en una arquitectura cliente-servidor. Sobre esta base, otro objetivo consiste en implementar una versión no interactiva mediante la transformación de Fiat-Shamir. Además, se propone incorporar una capa experimental de compromiso polinomial didáctico, junto con una fase de *trusted setup* simulado.

Por último, se plantea evaluar las distintas variantes implementadas y analizar sus limitaciones respecto a una prueba de conocimiento cero formal.

1.3. Estructura del trabajo

Este documento se organiza en cinco capítulos. En el Capítulo 2 se introducen los conceptos teóricos necesarios para comprender el trabajo, incluyendo las pruebas de conocimiento cero, las pruebas interactivas, el protocolo *sum-check*, la transformación de Fiat-Shamir y los compromisos polinomiales.

En el Capítulo 3 se describe el diseño e implementación del prototipo desarrollado. Se detalla la arquitectura cliente-servidor, la representación algebraica de la contraseña, el funcionamiento de las modalidades interactiva y no interactiva del protocolo, la capa experimental de compromisos polinomiales y el *trusted setup* simulado. También se recogen las configuraciones de autenticación disponibles y el despliegue del sistema mediante contenedores Docker.

En el Capítulo 4 se presentan las pruebas realizadas sobre el sistema, incluyendo la validación funcional, la evaluación de rendimiento y el número de peticiones intercambiadas entre cliente y servidor para cada configuración.

Finalmente, en el Capítulo 5 se exponen las conclusiones del trabajo, sus principales limitaciones y posibles líneas de continuación hacia construcciones criptográficas más completas, como compromisos polinomiales reales o sistemas zk-SNARK. Se incluye como anexo la planificación temporal y la dedicación asociada al desarrollo del proyecto.

Capítulo 2

Marco teórico

En este capítulo se introducen los conceptos teóricos necesarios para comprender el prototipo desarrollado. En primer lugar, se presentan las pruebas interactivas junto con sus propiedades básicas: completitud, solidez y ausencia de revelación de información adicional [1, 10]. Posteriormente, se detalla el contexto algebraico sobre el que se apoya el trabajo. En este marco aparecen las funciones definidas sobre el hipercubo booleano, las extensiones multilineales y los protocolos que verifican propiedades de polinomios mediante evaluaciones en puntos escogidos por el verificador [10]. Esta perspectiva permite presentar el protocolo *sum-check*, una prueba interactiva que verifica afirmaciones sobre sumas de evaluaciones de un polinomio [4, 10]. A continuación se describe la transformación de Fiat-Shamir, utilizada para obtener una versión no interactiva del protocolo [2, 10]. También se introducen los compromisos polinomiales y la idea de *trusted setup*, que en este trabajo se implementan de forma didáctica para mostrar la conexión con construcciones más avanzadas [5, 11]. Finalmente, se explica la relación de estos conceptos y herramientas con los zk-SNARKs utilizados en la actualidad en criptografía avanzada, y se delimitan las diferencias entre dichas construcciones y el prototipo implementado [10, 11].

2.1. Contexto criptográfico y algebraico

La clase de complejidad NP puede caracterizarse mediante relaciones para las que una instancia afirmativa dispone de un testigo verificable en tiempo polinómico. En este modelo, un algoritmo $\mathscr{P}$, denominado *prover*, proporciona el testigo o certificado para que otro algoritmo $\mathscr{V}$, denominado *verifier*, compruebe su validez. La comunicación se realiza en un único intercambio: el *prover* entrega el certificado y el *verifier* lo acepta o lo rechaza.

Goldwasser, Micali y Rackoff generalizaron este concepto mediante la definición de las pruebas interactivas [1]. En este nuevo modelo, la validez de una afirmación no necesariamente se comunica mediante una única prueba escrita, sino a través de una secuencia de mensajes entre $\mathscr{P}$ y $\mathscr{V}$. En este trabajo se utiliza la formulación moderna presentada por Thaler [10], ya que permite introducir de forma sencilla los protocolos algebraicos vistos más adelante.

Definición 2.1 (Prueba interactiva). [10] Sea $f : \{0,1\}^n \to \mathscr{R}$ una función con rango finito $\mathscr{R}$. Una prueba interactiva para f es un protocolo entre un *prover* $\mathscr{P}$ y un *verifier* $\mathscr{V}$, ambos con una entrada común $x \in \{0,1\}^n$. Al comienzo del protocolo, $\mathscr{P}$ proporciona un valor y, que afirma ser igual a $f(x)$. A continuación, $\mathscr{P}$ y $\mathscr{V}$ intercambian una secuencia de mensajes. Dicha secuencia, junto con el valor y, forma el registro de interacción del protocolo. Al finalizar la interacción, $\mathscr{V}$ acepta o rechaza la afirmación $y = f(x)$, representando el valor 1 la aceptación y el valor 0 el rechazo.

En este modelo, $\mathscr{V}$ se considera un algoritmo probabilista y de tiempo polinómico en n, mientras que $\mathscr{P}$ puede disponer de mayor capacidad computacional.

A partir de esta definición, se pueden introducir las propiedades principales de una prueba interactiva.

Denótese por

$$out(\mathscr{V}, x, y, r, \mathscr{P}) \in \{0,1\}$$

la salida del *verifier* al finalizar el protocolo, donde $x \in \{0,1\}^n$ es la entrada de $\mathscr{V}, \mathscr{P}$, y es el valor afirmado por el *prover*, r representa la aleatoriedad interna de $\mathscr{V}$, y $\mathscr{P}$ es la estrategia seguida por el *prover*. El valor 1 representa aceptación y el valor 0 representa rechazo.

Definición 2.2 (Error de completitud). [10] Una prueba interactiva $(\mathscr{P}, \mathscr{V})$ tiene error de completitud δ_c si, para toda entrada $x \in \{0,1\}^n$, cuando el valor afirmado por el *prover* es correcto, es decir, $y = f(x)$, se cumple que

$$\Pr_r[out(\mathscr{V}, x, y, r, \mathscr{P}) = 1] \geq 1 - \delta_c. \tag{2.1}$$

Es decir, si la afirmación es verdadera y el *prover* sigue honestamente el protocolo, el *verifier* acepta con probabilidad alta.

Definición 2.3 (Error de solidez). [10] Una prueba interactiva $(\mathscr{P}, \mathscr{V})$ tiene error de solidez δ_s si, para toda entrada $x \in \{0,1\}^n$, todo valor $y \neq f(x)$, y cualquier estrategia de un *prover* deshonesto $\mathscr{P}'$, se cumple que

$$\Pr_r\left[out(\mathscr{V}, x, y, r, \mathscr{P}') = 1\right] \leq \delta_s. \tag{2.2}$$

Es decir, si la afirmación es falsa, ningún *prover* deshonesto debería poder convencer al *verifier* de aceptarla salvo con probabilidad pequeña.

Definición 2.4 (Sistema de prueba válido). [10] Un sistema basado en una prueba interactiva $(\mathscr{P}, \mathscr{V})$ se considera válido si sus errores de completitud y solidez están acotados por una constante lo suficientemente pequeña. Habitualmente se considera

$$\delta_c, \delta_s \leq \frac{1}{3}. \tag{2.3}$$

Además de completitud y solidez, en criptografía resulta especialmente importante la propiedad de conocimiento cero. Esta propiedad exige que el *verifier* no aprenda información adicional sobre el secreto o testigo utilizado por el *prover*, más allá de la validez de la afirmación demostrada. Formalmente, se expresa mediante la existencia de un simulador capaz de reproducir la vista del *verifier* sin disponer del testigo secreto [1, 10].

Una vez introducidas las propiedades generales de las pruebas interactivas, se describe el contexto algebraico en el que se apoya el protocolo *sum-check*. Este protocolo trabaja con afirmaciones sobre polinomios definidos sobre un cuerpo finito. Por ello, antes de presentar el protocolo, es necesario introducir las funciones definidas sobre el hipercubo booleano y cómo estas se pueden extender a polinomios evaluables fuera de los puntos booleanos.

Definición 2.5 (Extensión polinomial). Sea $\mathbb{F}$ un cuerpo finito y sea $f : \{0,1\}^v \to \mathbb{F}$ una función que mapea el hipercubo booleano de dimensión v a escalares en $\mathbb{F}$. Un polinomio $g : \mathbb{F}^v \to \mathbb{F}$ en v variables se dice extensión de f si

$$g(x) = f(x), \quad \forall x \in \{0,1\}^v. \tag{2.4}$$

Como consecuencia del lema de Schwartz-Zippel [10], si dos extensiones polinomiales de bajo grado son distintas como polinomios, sólo pueden coincidir en una fracción pequeña de puntos del cuerpo, siempre que el grado sea mucho menor que $|\mathbb{F}|$. Esta idea conduce de forma natural a preguntarse si la extensión polinomial de una función definida sobre el hipercubo booleano es única. Sin embargo, esto no es cierto en general, y por ello se introduce la noción de extensión multilineal.

Definición 2.6 (**Polinomio multilineal**). Un polinomio multivariable $g : \mathbb{F}^v \to \mathbb{F}$ se dice multilineal si el grado del polinomio en cada variable es menor o igual que 1.

Proposición 2.1. *Toda función $f : \{0,1\}^v \to \mathbb{F}$ sobre el hipercubo booleano tiene extensión multilineal única sobre $\mathbb{F}$, la cual se denota mediante $\tilde{f}$.*

Esto es, $\tilde{f}$ es el único polinomio multilineal sobre $\mathbb{F}$ tal que

$$\tilde{f}(x) = f(x), \quad \forall x \in \{0,1\}^v.$$

Además, es posible hallar una expresión explícita para esta extensión multilineal mediante el siguiente resultado:

Lema 2.2 (**Interpolación de Lagrange**). *Sea $f : \{0,1\}^v \to \mathbb{F}$ una función definida sobre el hipercubo booleano, la extensión multilineal de f se puede hallar como*

$$\tilde{f}(x_1,\ldots,x_v) = \sum_{w\in\{0,1\}^v} f(w)\cdot\chi_w(x_1,\ldots,x_v) \tag{2.5}$$

donde, para cada $w = (w_1,\ldots,w_v)$, $w_i \in \{0,1\}$ $\forall i = 1,\ldots,v$ se tiene

$$\chi_w(x_1,\ldots,x_v) := \prod_{i=1}^{v}(x_i w_i + (1-x_i)(1-w_i)) \tag{2.6}$$

El conjunto $\{\chi_w \mid w \in \{0,1\}^v\}$ forma una base para representar funciones definidas sobre el hipercubo booleano mediante polinomios multilineales.

Demostración: Véase [10, pp. 29-30]. □

Esta unicidad de la extensión multilineal es el resultado principal que se precisa para introducir el protocolo *sum-check*. Además, si se dispone de la tabla completa de evaluaciones de f, dicha extensión puede evaluarse en un punto arbitrario $r \in \mathbb{F}^v$ en tiempo lineal en el tamaño de la tabla sin necesidad de hallar su expresión de forma explícita.

Lema 2.3. *Sea $v \in \mathbb{Z}^+$ y sea $n = 2^v$. Sea $f : \{0,1\}^v \to \mathbb{F}$ una función sobre el hipercubo booleano de dimensión v, y supongamos que se conocen todos los valores $f(w)$, para $w \in \{0,1\}^v$. Dado un vector $r = (r_1,\ldots,r_v) \in \mathbb{F}^v$, es posible calcular $\tilde{f}(r)$ en tiempo $O(n)$ y espacio $O(n)$.*

Demostración: Por la fórmula de interpolación de Lagrange multilineal, se tiene

$$\tilde{f}(r) = \sum_{w\in\{0,1\}^v} f(w)\cdot\chi_w(r). \tag{2.7}$$

Por tanto, basta con calcular los valores $\chi_w(r)$ para todos los puntos booleanos $w \in \{0,1\}^v$ y realizar después el producto escalar con la tabla de valores de f.

Para calcular estos valores $\chi_w(r)$ se puede usar un procedimiento incremental. Se define inicialmente una tabla $A^{(0)}$ formada por un único valor,

$$A^{(0)}[\emptyset] = 1. \tag{2.8}$$

En la etapa j, con $1 \le j \le v$, se construye una tabla $A^{(j)}$ con 2^j entradas, una por cada $(w_1,\ldots,w_j) \in \{0,1\}^j$, mediante la relación

$$A^{(j)}[w_1,\ldots,w_j] = A^{(j-1)}[w_1,\ldots,w_{j-1}]\cdot(w_j r_j + (1-w_j)(1-r_j)). \tag{2.9}$$

Al finalizar la etapa v, la tabla $A^{(v)}$ contiene exactamente los valores $\chi_w(r)$ para todos los puntos $w \in \{0,1\}^v$. Cada etapa j requiere tiempo proporcional a 2^j, por lo que el coste total es

$$\sum_{j=1}^{v} 2^j = O(2^v) = O(n). \tag{2.10}$$

La tabla final tiene tamaño n, por lo que el espacio utilizado también es $O(n)$. Finalmente, el valor $\tilde{f}(r)$ se obtiene calculando

$$\tilde{f}(r) = \sum_{w \in \{0,1\}^v} f(w) \cdot A^{(v)}[w]. \tag{2.11}$$

Por tanto, $\tilde{f}(r)$ puede calcularse en tiempo y espacio lineales en el tamaño de la tabla de evaluaciones de f. □

Esta forma de evaluar la extensión multilineal será útil en el protocolo *sum-check*. Dicho protocolo no verifica directamente todos los valores de una función sobre el hipercubo booleano, sino que reduce progresivamente una afirmación sobre una suma de evaluaciones a una comprobación final en un punto aleatorio del cuerpo finito. Por tanto, las extensiones multilineales proporcionan el puente entre una función definida sobre $\{0,1\}^v$ y las evaluaciones algebraicas que aparecen durante la interacción entre *prover* y *verifier*.

2.2. El protocolo *sum-check*

El protocolo *sum-check* tiene su origen en las técnicas algebraicas introducidas por Lund, Fortnow, Karloff y Nisan para la construcción de pruebas interactivas [4]. En dicho trabajo, los autores muestran cómo utilizar polinomios de bajo grado y evaluaciones aleatorias para transformar la verificación de afirmaciones globales en una secuencia de comprobaciones locales.

En este trabajo se utilizará la formulación moderna del protocolo presentada por Thaler [10], donde *sum-check* se describe directamente como un protocolo para verificar afirmaciones de la forma

$$H = \sum_{b \in \{0,1\}^v} g(b), \tag{2.12}$$

siendo $g : \mathbb{F}^v \to \mathbb{F}$ un polinomio en v variables sobre un cuerpo finito $\mathbb{F}$.

Nótese en primer lugar que, si el *verifier* decidiera realizar una comprobación directa de esta igualdad, tendría que evaluar g en los 2^v puntos del hipercubo booleano, lo que resulta costoso cuando v crece. El protocolo *sum-check* evita esta comprobación exhaustiva mediante una interacción de v rondas, reduciendo en cada ronda la afirmación pendiente a otra afirmación con una variable menos.

El funcionamiento del protocolo *sum-check* puede resumirse mediante el Algoritmo 1. C_j representa el valor que el *verifier* debe comprobar en la ronda j. Inicialmente, C_1 es el valor afirmado para la suma total. En cada ronda, el *prover* envía un polinomio univariante g_j y el *verifier* comprueba que

$$g_j(0) + g_j(1) = C_j. \tag{2.13}$$

Si la comprobación se cumple, el reto aleatorio r_j fija la variable X_j y permite pasar a la siguiente ronda con el valor $C_{j+1} = g_j(r_j)$. De este modo, el protocolo va plegando la suma sobre el hipercubo booleano: de la suma inicial sobre 2^v puntos se pasa a una suma sobre 2^{v-1} puntos, después sobre 2^{v-2}, y así sucesivamente hasta reducir la comprobación a un único valor.

Algoritmo 1 Protocolo *sum-check*

Require: Polinomio $g : \mathbb{F}^v \to \mathbb{F}$ y cotas $\deg_j(g)$ para cada variable X_j.
Require: Valor inicial C_1, afirmado por el *prover* como igual a la suma $\sum_{b\in\{0,1\}^v} g(b)$.
Ensure: Aceptar o rechazar la afirmación.

for $j = 1, \ldots, v$ **do**
    El *prover* envía un polinomio univariante $g_j(X_j)$.
    El *verifier* comprueba que $\deg(g_j) \leq \deg_j(g)$.
    El *verifier* comprueba que $g_j(0) + g_j(1) = C_j$.
    if alguna comprobación falla **then**
        return Rechazar.
    end if
    El *verifier* escoge un reto aleatorio $r_j \in \mathbb{F}$.
    El *verifier* actualiza $C_{j+1} = g_j(r_j)$.
end for
El *verifier* evalúa $g(r_1, \ldots, r_v)$.
if $C_{v+1} = g(r_1, \ldots, r_v)$ **then**
    return Aceptar.
else
    return Rechazar.
end if

El polinomio que debería enviar un *prover* honesto en la ronda j es

$$g_j(X_j) = \sum_{x_{j+1},\ldots,x_v\in\{0,1\}} g(r_1, \ldots, r_{j-1}, X_j, x_{j+1}, \ldots, x_v). \tag{2.14}$$

Para $j = 1$, esta expresión no contiene retos anteriores. Tras la última ronda, todas las variables han quedado fijadas a los retos $r_1, \ldots, r_v$, por lo que el *verifier* solo tiene que comprobar

$$C_{v+1} = g(r_1, \ldots, r_v). \tag{2.15}$$

Una vez descrito el funcionamiento del protocolo, conviene relacionarlo con las propiedades de una prueba interactiva introducidas al comienzo del capítulo.

Proposición 2.4 (Completitud y solidez de *sum-check*). *Sea g un polinomio en v variables sobre un cuerpo finito $\mathbb{F}$, con grado a lo sumo d en cada variable. El protocolo sum-check tiene error de completitud $\delta_c = 0$ y error de solidez acotado por*

$$\delta_s \leq \frac{vd}{|\mathbb{F}|}. \tag{2.16}$$

Demostración: Véase [10, pp. 36-37]. □

Se tiene completitud ya que si el *prover* sigue honestamente el protocolo, todos los polinomios enviados coinciden con las sumas parciales correspondientes. La solidez en cambio se apoya en que, si un *prover* deshonesto envía en alguna ronda un polinomio distinto del correcto, ambos polinomios solo pueden coincidir en un número limitado de puntos. Como el reto del *verifier* se elige aleatoriamente en $\mathbb{F}$, la probabilidad de que un polinomio incorrecto pase la comprobación es pequeña cuando $|\mathbb{F}|$ es suficientemente grande.

También resulta relevante observar qué información necesita realmente el *verifier* para ejecutar el protocolo. Los mensajes enviados a modo de reto por parte del *verifier* al *prover* son únicamente escalares aleatorios del cuerpo finito $\mathbb{F}$, por lo que el *verifier* no necesita conocer de forma explícita el polinomio g. Para modelar el protocolo, basta con conocer una cota para el grado de g en cada variable y poder evaluar g en

el punto final $(r_1, \ldots, r_v)$ [10]. Esta propiedad permite que *sum-check* pueda utilizarse como componente dentro de protocolos algebraicos más complejos.

Finalmente, se puede analizar cómo de efectivo resulta *sum-check* a la hora de reducir el coste. El protocolo realiza una ronda por cada variable del polinomio g. El *verifier* envía un reto aleatorio en cada ronda al *prover*, es decir, v elementos de $\mathbb{F}$. En sentido inverso, el *prover* envía en cada ronda un polinomio univariante g_j, que puede representarse mediante $\deg_j(g)+1$ coeficientes. Por tanto, la comunicación del *prover* está ligada a $\sum_{j=1}^{v}(\deg_j(g)+1)$ y, cuando el grado en cada variable está acotado por una constante, su coste es lineal en v.

La ventaja principal aparece en el coste del *verifier*, ya que en lugar de evaluar g en los 2^v puntos del hipercubo booleano, solo realiza comprobaciones sobre polinomios univariantes y una evaluación final de g en el punto aleatorio $(r_1, \ldots, r_v)$. Por ello, su tiempo de ejecución es proporcional al coste de comunicación, más el coste de dicha evaluación final [10].

El protocolo básico descrito garantiza completitud y solidez, pero no proporciona por sí mismo la propiedad de conocimiento cero. La obtención de una variante de *sum-check* con dicha propiedad requiere mecanismos adicionales que oculten la información revelada por los polinomios de ronda. Existen construcciones específicas de *sum-check* de conocimiento cero [7]; no obstante, el prototipo desarrollado en este trabajo implementa la versión algebraica básica y no incorpora dichas técnicas de ocultación.

2.3. La transformación de Fiat-Shamir

El protocolo *sum-check* descrito en la sección anterior constituye una prueba interactiva: en cada ronda, el *verifier* escoge un reto aleatorio y lo envía al *prover*. Esta comunicación permite introducir aleatoriedad impredecible durante la ejecución del protocolo, dificultando que un *prover* deshonesto consiga que el *verifier* acepte una afirmación falsa.

Sin embargo, en muchas aplicaciones resulta conveniente eliminar esta interacción y obtener una prueba que pueda generarse y verificarse sin mantener una comunicación ronda a ronda. En este contexto surge la transformación de Fiat-Shamir, introducida originalmente en el ámbito de los esquemas de identificación y firma digital [2]. Su idea principal es convertir ciertos protocolos interactivos basados en retos y respuestas en protocolos no interactivos.

Para ello, los retos aleatorios enviados por el *verifier* se sustituyen por valores calculados mediante una función *hash* aplicada al registro de interacción parcial del protocolo. Así, en lugar de recibir un reto externo, el *prover* calcula dicho reto de forma determinista a partir de los mensajes generados hasta ese momento.

Aplicada al protocolo *sum-check*, esta transformación reemplaza cada reto aleatorio $r_j \in \mathbb{F}$ por un valor derivado del registro de interacción entre *prover* y *verifier*:

$$r_j = H(\mathsf{tr}_j). \tag{2.17}$$

Dicho registro tr_j debe incluir la información pública que determina la instancia del protocolo y los mensajes producidos hasta la ronda j. En particular, debe recoger la afirmación que se desea verificar, el valor inicial y los mensajes previos enviados por el *prover*. Incluir la información pública de la instancia es importante para evitar que un adversario pueda generar una prueba y adaptarla posteriormente a otra afirmación distinta (ataque de retransmisión) [10].

En la Figura 2.1, tr_j representa el registro parcial hasta la ronda j. En la versión interactiva, los retos r_j son escogidos por el *verifier*; en la versión no interactiva, sin embargo, se derivan aplicando una función *hash* a dicho registro. De este modo, cada reto queda ligado tanto a la instancia concreta del protocolo como a los mensajes anteriores.

Fiat-Shamir no modifica las comprobaciones algebraicas de *sum-check*; únicamente sustituye la forma en que se obtienen los retos aleatorios. El *prover* puede generar de una sola vez los mensajes que formarían

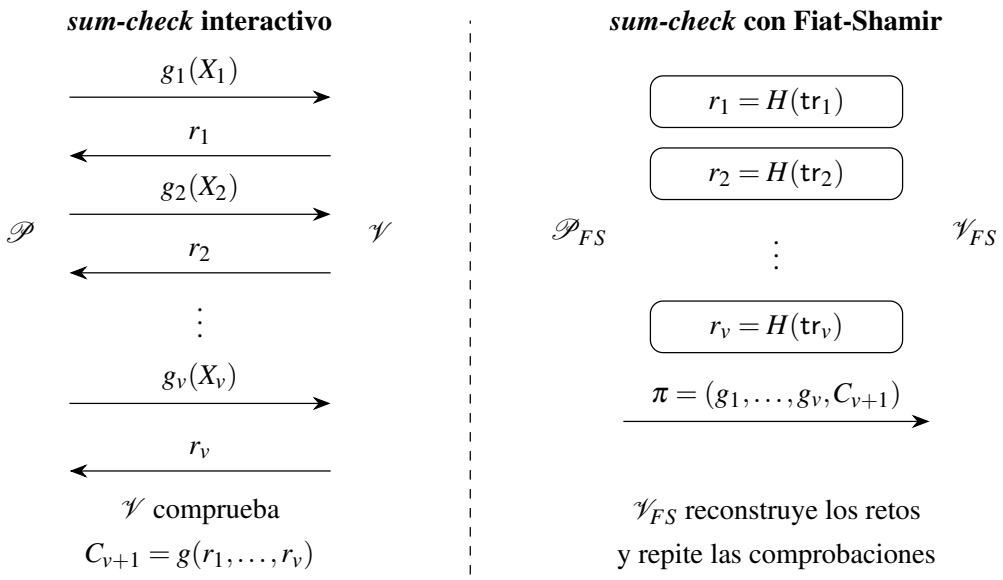

Figura 2.1: Comparación entre *sum-check* interactivo y su versión no interactiva mediante Fiat-Shamir. Inspirado en [10, pp. 76]

.

parte de la ejecución interactiva, mientras que el *verifier* reconstruye los retos y ejecuta las mismas comprobaciones de consistencia del protocolo original.

El análisis formal de la transformación de Fiat-Shamir suele realizarse en el modelo de oráculo aleatorio [3]. En este modelo, la función *hash* se idealiza como un oráculo que, para cada entrada nueva, devuelve un valor aleatorio uniforme, manteniendo siempre la misma respuesta si se repite la misma consulta. Bajo esta premisa, y para las familias de protocolos que satisfacen las condiciones requeridas por la transformación, los retos derivados del registro de interacción pueden tratarse como si hubieran sido escogidos aleatoriamente por el *verifier*.

2.4. Compromisos polinomiales y *trusted setup*

Un compromiso criptográfico permite fijar un valor para impedir que se modifique posteriormente. En primer lugar, se publica un compromiso, que actúa como un dato asociado al mensaje original. Más adelante, ese compromiso puede *abrirse*, revelando el mensaje subyacente y la información necesaria para comprobar que coincide con el compromiso publicado inicialmente. En los esquemas de compromiso generales se distinguen habitualmente las propiedades de ocultación y vinculación. En un esquema de compromiso polinomial resulta esencial que no puedan producirse aperturas incompatibles para el polinomio comprometido; la propiedad de ocultación puede requerir mecanismos adicionales dependiendo de la construcción [5].

Los compromisos polinomiales trasladan esta idea al caso de los polinomios. En lugar de comprometer un mensaje aislado, se compromete un polinomio $f(X)$. El *prover* publica un compromiso compacto asociado a dicho polinomio y, posteriormente, puede abrirlo en un punto concreto para demostrar que el valor declarado coincide con la evaluación de f en ese punto, sin tener que revelar necesariamente todos sus coeficientes [5, 11].

De forma abstracta, un esquema de compromiso polinomial puede describirse mediante tres operaciones principales. La operación de compromiso genera un valor C_f asociado al polinomio $f(X)$. La operación de apertura, aplicada a un punto z, produce el valor evaluado $y = f(z)$ junto con una prueba π de que dicha evaluación es coherente con el polinomio comprometido. Finalmente, la operación de verificación

comprueba, a partir de C_f, z, y y π, si la apertura debe aceptarse.

$$C_f \leftarrow \text{Commit}(f), \tag{2.18}$$
$$(y, \pi) \leftarrow \text{Open}(f, z), \tag{2.19}$$
$$b \leftarrow \text{Verify}(C_f, z, y, \pi), \tag{2.20}$$

$b \in \{0,1\}$ indica si la apertura es aceptada o rechazada.

En este tipo de esquema, las operaciones de compromiso y apertura son ejecutadas por el *prover*, que conoce el polinomio $f(X)$. La apertura producida se proporciona al *verifier*, que ejecuta la operación de verificación usando el compromiso C_f, el punto de apertura z, el valor declarado y y la prueba π.

Una construcción especialmente relevante dentro de los compromisos polinomiales es la propuesta por Kate, Zaverucha y Goldberg, conocida habitualmente como compromiso KZG [5]. Su idea principal consiste en asociar el compromiso de un polinomio con una evaluación en un punto secreto.

Si el polinomio subyacente es

$$f(X) = a_0 + a_1 X + \cdots + a_d X^d, \tag{2.21}$$

el compromiso puede interpretarse de forma intuitiva como un valor relacionado con $f(\tau)$, donde τ es un escalar *secreto* elegido durante la fase de configuración. En un esquema KZG real, este valor no se publica directamente. La fase de configuración genera una cadena de referencia estructurada (o SRS, por sus siglas en inglés) que contiene potencias de τ codificadas en un grupo criptográfico. Gracias a esta cadena, *prover* y *verifier* pueden calcular compromisos y verificar aperturas sin conocer directamente el valor secreto τ [5, 11].

La apertura de un compromiso en un punto z se basa en una identidad elemental de divisibilidad de polinomios. Si el valor declarado por el *prover* es $y = f(z)$, entonces el polinomio $f(X) - y$ se anula en $X = z$. Por tanto, $X - z$ divide a $f(X) - y$, y existe un polinomio cociente

$$q(X) = \frac{f(X) - y}{X - z}. \tag{2.22}$$

Evaluando formalmente esta relación en el punto secreto τ, se obtiene

$$f(\tau) - y = (\tau - z) q(\tau). \tag{2.23}$$

Esta igualdad sirve como base algebraica de la verificación de aperturas: el *prover* proporciona una prueba asociada al cociente $q(X)$, y el *verifier* comprueba que dicha prueba es compatible con el compromiso, el punto de apertura y el valor declarado [5, 11].

La seguridad del esquema depende por tanto de que el valor secreto utilizado para generar la SRS no sea conocido por las partes una vez finalizada la configuración. Por este motivo, esta fase se denomina *trusted setup*: si el secreto no se destruye o se vuelve público, las garantías de vinculación del compromiso pueden verse comprometidas. En este sentido, la SRS actúa como un conjunto de parámetros públicos que permite operar con evaluaciones relacionadas con $f(\tau)$, pero sin revelar directamente el punto secreto τ [5, 11].

Los compromisos polinomiales proporcionan, por tanto, una herramienta para sustituir polinomios completos por compromisos y aperturas verificables. Esta idea es la base que permite relacionar los protocolos algebraicos anteriores con las construcciones zk-SNARK en este trabajo.

2.5. Relación con construcciones zk-SNARK

Las secciones anteriores han introducido varias herramientas que aparecen en muchas construcciones modernas de conocimiento cero: protocolos algebraicos como *sum-check*, transformaciones para eliminar

interacción como Fiat-Shamir y compromisos polinomiales para trabajar con polinomios sin revelarlos de forma explícita. Estas herramientas no forman, por sí solas, un zk-SNARK completo, pero sí permiten entender algunas de las piezas que intervienen en este tipo de sistemas.

El término zk-SNARK proviene de *Zero-Knowledge Succinct Non-Interactive Argument of Knowledge*. Como se mencionó al inicio del capítulo, la propiedad de conocimiento cero indica que la prueba no revela información adicional sobre el testigo privado. La sucintez hace referencia a que tanto la prueba como su verificación son pequeñas en comparación con el cálculo original. La no interacción permite que la prueba pueda generarse y verificarse en un único intercambio, sin necesidad de una conversación ronda a ronda. Finalmente, el término argumento de conocimiento expresa que la prueba es suficiente para convencer al *verifier* de que el *prover* conoce un testigo válido para la afirmación considerada [11, 12].

El desarrollo de un zk-SNARK suele organizarse en varias etapas. En primer lugar, la afirmación que se quiere probar se transforma en una representación algebraica, por ejemplo mediante un circuito aritmético o sistemas de restricciones. Este paso se conoce como aritmetización. A partir de esa representación, se plantean relaciones que deben cumplirse sobre los objetos resultantes, normalmente restricciones o identidades polinomiales. Protocolos como *sum-check* muestran cómo este tipo de comprobaciones puede reducirse a razonamientos algebraicos sobre polinomios. La parte criptográfica utiliza herramientas como compromisos polinomiales y, en muchos casos, la transformación de Fiat-Shamir para obtener una prueba no interactiva y verificable de forma eficiente [11].

Los compromisos polinomiales cumplen un papel intermedio entre la descripción algebraica del cálculo y la prueba criptográfica final. En lugar de enviar todos los polinomios que aparecen en la aritmetización, el *prover* puede enviar compromisos a esos polinomios y abrirlos en los puntos que sean necesarios. El *verifier* comprueba entonces esas aperturas y las relaciones algebraicas correspondientes sin manejar directamente todos los datos subyacentes.

Aun así, disponer de protocolos algebraicos, Fiat-Shamir y compromisos polinomiales no basta para obtener automáticamente un zk-SNARK completo. Una construcción de este tipo requiere concretar la relación que se quiere probar, definir su aritmetización, establecer qué restricciones o identidades polinomiales deben verificarse, justificar el protocolo algebraico utilizado y escoger el mecanismo criptográfico que convierte todo ello en una prueba sucinta, no interactiva y segura. Por ello, las herramientas estudiadas en este capítulo deben entenderse como componentes de una arquitectura más amplia, no como una construcción completa por sí mismas.

En la práctica existen distintas familias de sistemas ZK, cada una con compromisos diferentes entre tamaño de prueba, coste de verificación, supuestos criptográficos y necesidad de configuración inicial. Groth16 es un zk-SNARK de preprocesamiento basado en programas aritméticos cuadráticos y emparejamientos bilineales; destaca por producir pruebas pequeñas y verificaciones rápidas, aunque requiere una configuración inicial asociada al circuito [6]. Otras familias presentan una separación más explícita entre el protocolo algebraico y el esquema de compromiso polinomial utilizado.

Los zk-STARKs, en cambio, emplean técnicas transparentes y evitan el *trusted setup*, a costa de producir habitualmente pruebas de mayor tamaño [8, 12]. Sistemas como Bulletproofs tampoco requieren configuración de confianza y se han utilizado especialmente en pruebas de rango, con un tamaño de prueba logarítmico, aunque presentan otros costes de generación y verificación [9, 12].

Por tanto, los zk-SNARKs se sitúan como construcciones que combinan técnicas algebraicas y criptográficas: aritmetización de afirmaciones, comprobaciones sobre polinomios, compromisos polinomiales y eliminación de interacción. Recuérdese que las herramientas estudiadas en este capítulo no forman un zk-SNARK completo, pero ayudan a comprender por qué este tipo de sistemas se apoya en polinomios, compromisos y transformaciones no interactivas.

Capítulo 3

Diseño e implementación del sistema

En este capítulo se describe el diseño y la implementación del prototipo desarrollado. En primer lugar, se presenta su arquitectura general y se delimita la responsabilidad de los componentes ejecutados en el navegador, el servidor y la capa de persistencia. A continuación, se resume el flujo funcional del sistema, distinguiendo el registro de usuarios y las variantes de autenticación implementadas. Posteriormente, se detalla la implementación de los mecanismos algebraicos de autenticación. Para ello se describe la representación algebraica de la contraseña, la generación de los datos de registro, la autenticación interactiva mediante *sum-check*, la variante no interactiva basada en Fiat-Shamir y la extensión experimental de compromiso polinomial. Finalmente, se recogen las configuraciones disponibles y se describe el despliegue contenerizado utilizado para facilitar la reproducción del prototipo.

3.1. Arquitectura general

El prototipo se ha implementado como una aplicación web con arquitectura cliente-servidor y persistencia en PostgreSQL. La interacción con el usuario y la construcción de las pruebas se realizan en el navegador, mientras que el servidor gestiona los datos de verificación asociados a los usuarios. La comunicación entre ambas partes se lleva a cabo mediante peticiones HTTP cuyos datos se codifican en formato JSON.

La implementación se ha realizado principalmente en JavaScript. En el cliente se utilizan documentos HTML, una hoja de estilos CSS y módulos JavaScript ejecutados por el navegador. En el servidor se emplean Node.js como entorno de ejecución y Express para definir la API HTTP y servir los recursos de la aplicación. La información necesaria para verificar autenticaciones posteriores se conserva en una base de datos PostgreSQL.

La Figura 3.1 describe la arquitectura general del prototipo. Se distinguen cuatro bloques principales: el cliente web, el servidor HTTP, los módulos que implementan el protocolo y la capa de persistencia. La parte cliente se encuentra en el directorio `public/`. Las vistas `index.html`, `register.html` y `private.html` corresponden a la página de inicio de sesión, el formulario de registro y la zona privada respectivamente. La presentación se define en `styles.css`, mientras que la gestión de los eventos y de las peticiones al servidor se realiza desde `login-app.js` y `register-app.js`.

Desde estos controladores se obtienen las credenciales introducidas por el usuario y se invocan los módulos necesarios para construir las pruebas. La contraseña se transforma en el propio navegador y no se incluye en las peticiones enviadas al servidor. Por tanto, el cliente adopta el papel de *prover* y transmite únicamente los mensajes y valores algebraicos derivados que requiere cada operación.

El servidor se organiza en el directorio `server/`. El fichero `server.js` constituye el punto de entrada de la aplicación y pone el servidor a la escucha en el puerto configurado. La configuración de Express, las rutas de la API, el servicio de los recursos estáticos y la coordinación de los procesos de registro y autenticación se concentran en `app.js`.

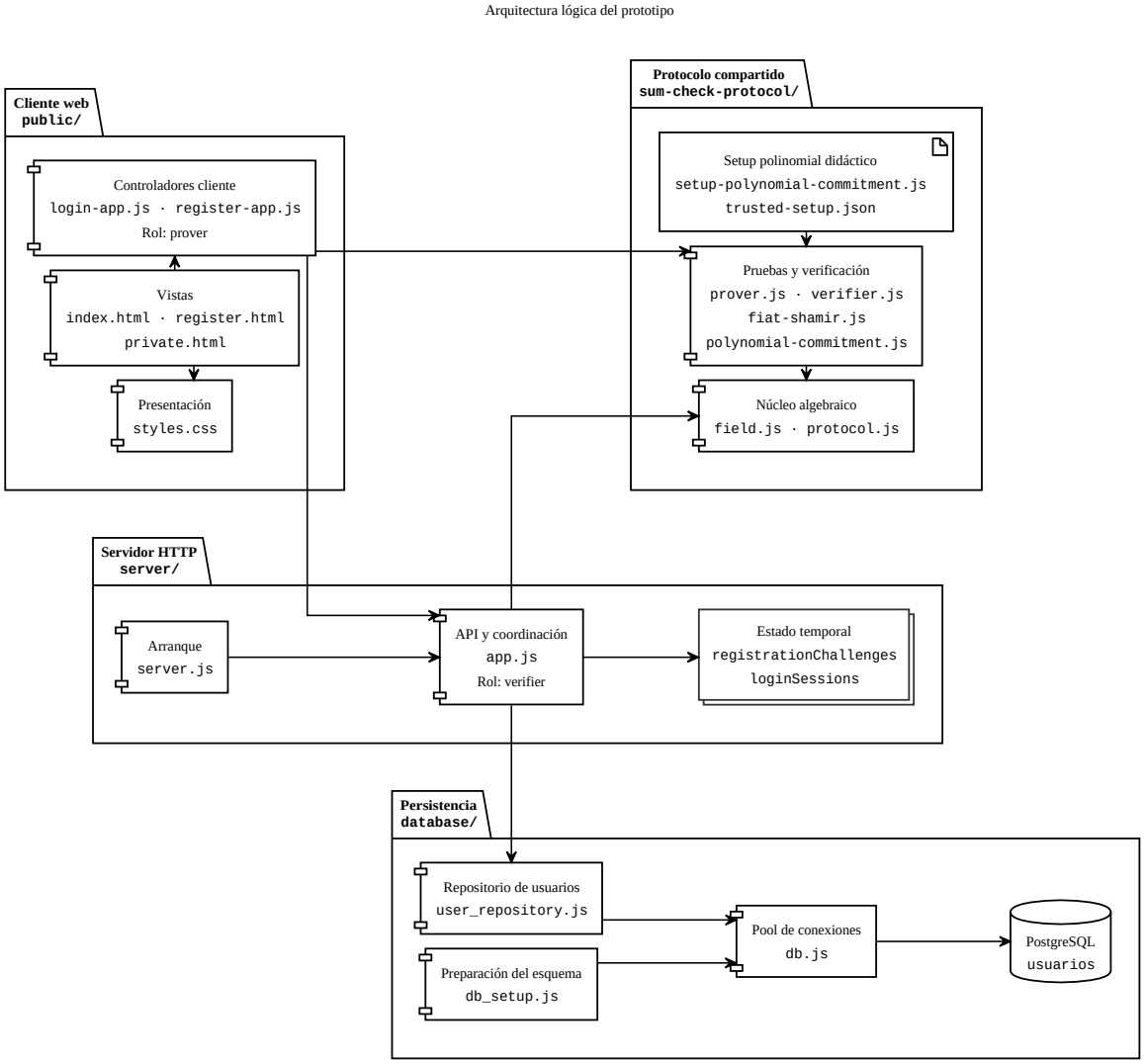

Figura 3.1: Arquitectura lógica del prototipo desarrollado.

Los intercambios interactivos requieren conservar información entre varias peticiones HTTP. Para ello se utilizan las estructuras en memoria `registrationChallenges` y `loginSessions`. La primera mantiene temporalmente los desafíos generados durante el registro, mientras que la segunda almacena las sesiones asociadas a los intentos de autenticación interactiva por parte de los usuarios. Sus entradas tienen una duración limitada y se eliminan cuando expiran o cuando finaliza el proceso correspondiente.

Cuando una autenticación resulta válida, se genera una *cookie* firmada que identifica la sesión web del usuario. Esta *cookie* se comprueba antes de servir la zona privada y permanece válida durante un periodo máximo de ocho horas. El token es autocontenido y no se conserva como una sesión persistente en la base de datos, a diferencia del estado temporal en memoria utilizado durante el protocolo.

La lógica algebraica se agrupa en el directorio `sum-check-protocol/`. Las operaciones sobre el cuerpo finito y las funciones comunes se implementan en `field.js` y `protocol.js`. Los componentes correspondientes a los papeles de *prover* y *verifier* se encuentran en `prover.js` y `verifier.js`, respectivamente. La transformación no interactiva se implementa en `fiat-shamir.js`, mientras que `polynomial-commitment.js` contiene la extensión experimental de compromiso polinomial.

Estos módulos se reutilizan en ambos extremos de la aplicación. Los controladores del navegador importan la lógica necesaria para construir las pruebas, mientras que el servidor utiliza las funciones y clases encargadas de verificarlas. De este modo, las operaciones algebraicas se mantienen separadas de la gestión de las rutas HTTP y de la interfaz de usuario.

La extensión polinomial utiliza además la configuración almacenada en `trusted-setup.json`. Este mecanismo se ha incorporado con fines didácticos para representar las etapas de compromiso, apertura y verificación. Dado que el valor utilizado como *trusted setup* es conocido por el prototipo, este diseño no proporciona las garantías criptográficas de un esquema KZG real.

La persistencia de los valores algebraicos necesarios se gestiona en el directorio `database/`. El fichero `db.js` configura y exporta el *pool* de conexiones con PostgreSQL. El script `db_setup.js` crea la tabla `usuarios` y añade las columnas que puedan faltar cuando se parte de una versión anterior del esquema. Por último, `user_repository.js` encapsula las operaciones de inserción y consulta utilizadas por la API.

En la tabla `usuarios` no se almacena la contraseña. Se conservan el nombre de usuario, el valor inicial, los valores finales esperados para las modalidades interactiva y Fiat-Shamir, los pesos utilizados, el punto de reto del modo interactivo, la longitud de la representación y el compromiso polinomial generado durante el registro. La opción disponible en la interfaz determina posteriormente si se verifica también una apertura de ese compromiso durante la autenticación.

Los pesos y los retos del protocolo se comunican al cliente cuando resultan necesarios para construir una prueba. En cambio, los valores finales esperados permanecen en el servidor y se utilizan durante la verificación. La arquitectura separa así la construcción de las pruebas, realizada en el navegador, de su verificación y persistencia en el servidor.

La Figura 3.2 resume el funcionamiento del prototipo, sin entrar en los detalles de implementación de cada funcionalidad desarrollada. Sobre la arquitectura anterior se implementan dos operaciones principales: el registro y la autenticación. Durante el registro, el servidor genera los parámetros necesarios para preparar una cuenta. A partir de la contraseña y de estos parámetros, el navegador calcula los valores necesarios para verificar posteriores intentos de autenticación, y estos se almacenan en la tabla `usuarios`.

Durante la autenticación se recuperan los datos mencionados y se ejecuta una de las dos modalidades disponibles. En el modo interactivo se intercambian varias rondas entre el cliente y el servidor. En el modo no interactivo se construye una prueba completa mediante la transformación de Fiat-Shamir y se envía para su verificación en una única petición.

En ambos casos se puede incluir adicionalmente la comprobación de una apertura polinomial. Si se selecciona, la apertura generada por el navegador se contrasta con el compromiso almacenado durante el registro. La autenticación solo se acepta si se superan las comprobaciones correspondientes. Cuando

el resultado es válido, se genera una *cookie* firmada y se permite el acceso a la zona privada. Si alguna comprobación falla, se rechaza el intento y se elimina el estado temporal que se hubiera creado.

3.2. Implementación de la autenticación algebraica

El núcleo del prototipo se encuentra en el directorio `sum-check-protocol/`. El fichero `field.js` implementa las operaciones sobre el cuerpo finito; `protocol.js` contiene la construcción y el plegado de las tablas de evaluaciones; `prover.js` mantiene el estado del *prover* interactivo; y `verifier.js` implementa el estado y las comprobaciones realizadas por el *verifier*. Sobre estos componentes se construyen la variante no interactiva de `fiat-shamir.js` y la extensión de compromiso polinomial de `polynomial-commitment.js`.

3.2.1. Representación algebraica de la contraseña

Todas las operaciones algebraicas se realizan sobre el cuerpo primo

$$\mathbb{F}_p, \qquad p = 18446744073709551557. \tag{3.1}$$

Los elementos del cuerpo se representan mediante el tipo `BigInt` de JavaScript y se normalizan módulo p después de cada operación. Como JSON no permite serializar directamente este tipo, los elementos del campo se transmiten y almacenan como cadenas decimales.

Durante el registro se selecciona una longitud de contraseña de d bytes. El prototipo admite tamaños entre 4 y 256 bytes que sean potencias de dos. La contraseña debe estar formada por caracteres ASCII y tener exactamente la longitud escogida. Cada byte se descompone en sus ocho bits, comenzando por el bit más significativo, obteniéndose un vector

$$\mathbf{b} = (b_0, \ldots, b_{N-1}), \qquad N = 8d, \tag{3.2}$$

donde $b_i \in \{0,1\}$. Como N es potencia de dos, puede interpretarse como el número de evaluaciones de una función definida sobre un hipercubo booleano de dimensión

$$v = \log_2 N. \tag{3.3}$$

El servidor genera un vector de pesos públicos $\mathbf{w} = (w_0, \ldots, w_{N-1})$, cuyos elementos son valores aleatorios no nulos de $\mathbb{F}_p$. El cliente combina cada bit de la contraseña con su peso correspondiente:

$$g_i = b_i w_i. \tag{3.4}$$

El vector $(g_0, \ldots, g_{N-1})$ se interpreta como la tabla de evaluaciones de una función $g : \{0,1\}^v \to \mathbb{F}_p$, y el valor inicial se calcula como

$$C_0 = \sum_{i=0}^{N-1} g_i = \sum_{x \in \{0,1\}^v} g(x). \tag{3.5}$$

Este valor constituye el punto de partida de las verificaciones posteriores. La notación comienza en C_0 para ajustarse a la indexación utilizada por la implementación JavaScript. Corresponde al valor C_1 empleado como afirmación inicial en la descripción teórica del Algoritmo 1.

3.2.2. Generación de los datos de registro

El registro comienza mediante una petición `GET/api/zkp/register-challenge`. El servidor valida el tamaño solicitado, genera los pesos $\mathbf{w}$ y un punto $\mathbf{r} = (r_1, \ldots, r_v)$, y asocia estos parámetros a un

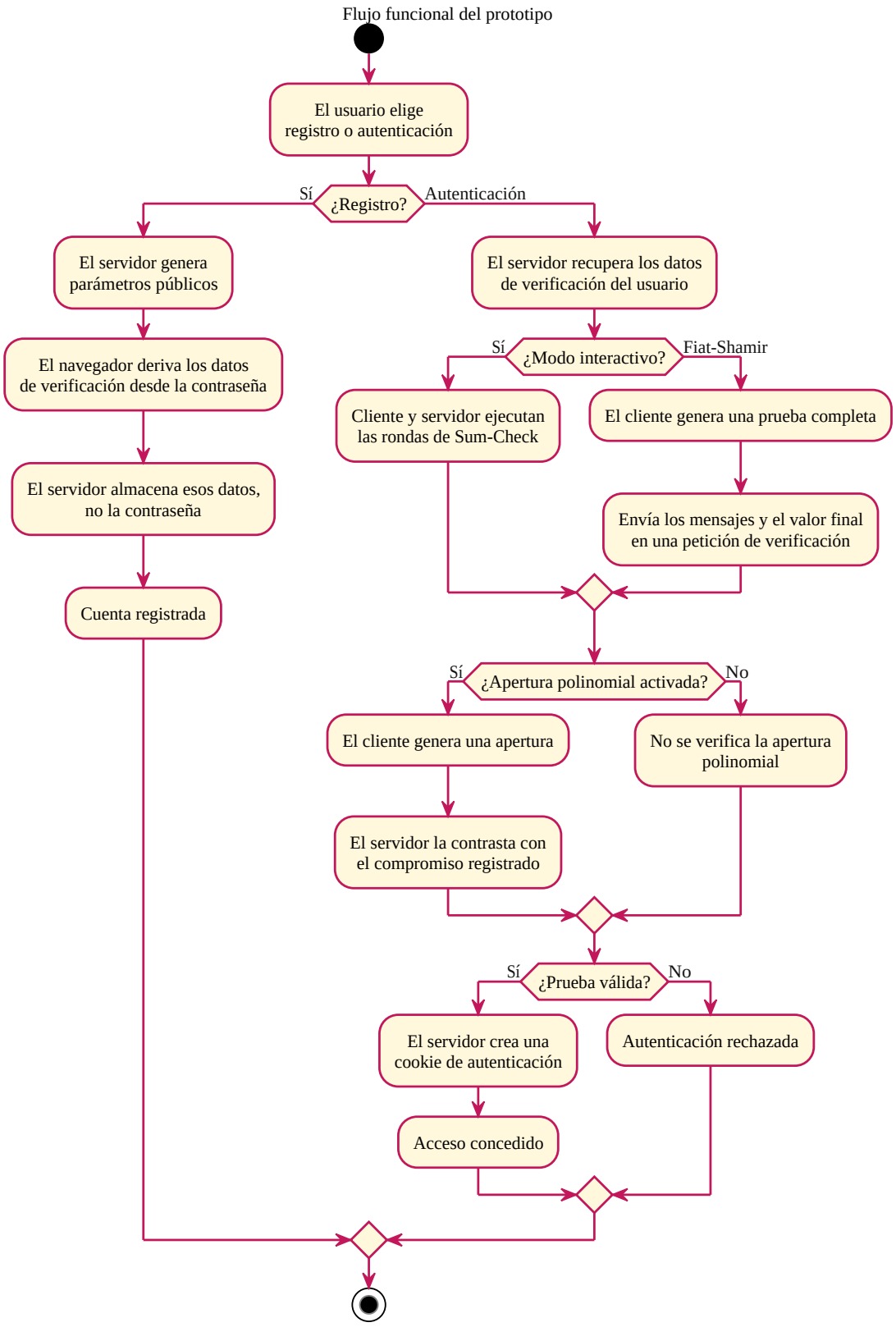

Figura 3.2: Flujo funcional general del prototipo.

identificador temporal `challengeId`. La entrada permanece en el mapa `registrationChallenges` hasta que se completa el registro o expira su periodo de validez.

El navegador construye un objeto `InteractiveSumCheckProver` a partir de la contraseña, los pesos y el punto recibido. Este objeto conserva la tabla inicial de evaluaciones y calcula C_0. A continuación pliega localmente la tabla utilizando las coordenadas de $\mathbf{r}$ hasta obtener el valor final de la modalidad interactiva, $g(\mathbf{r})$.

El cliente también ejecuta localmente la construcción Fiat-Shamir para obtener el valor final correspondiente a los retos derivados del registro. Ambos cálculos deben producir el mismo valor inicial, aunque sus valores finales son generalmente distintos porque utilizan retos diferentes.

Finalmente, el navegador obtiene el compromiso polinomial y envía, mediante la petición `POST/api/zkp/register` el nombre de usuario, C_0, el valor final interactivo, el valor final Fiat-Shamir y el compromiso. Para concluir, el servidor añade los pesos, el punto interactivo y la longitud.

En consecuencia, la tabla `usuarios` almacena `username`, `initialClaim`, `final_value`, `final_value_fs`, `weights_json`, `challenge_point_json`, `bit_length` y `poly_commitment`. La contraseña y su representación binaria no se incluyen en las peticiones ni se almacenan.

Esta operación debe entenderse como una fase de enrolamiento. El servidor no reconstruye la contraseña ni verifica de forma independiente durante el registro que los valores recibidos procedan de una contraseña concreta. En su lugar, vincula la cuenta con los datos algebraicos proporcionados por el cliente, que se utilizan como referencia en autenticaciones posteriores. La Figura 3.3 muestra el proceso de registro mediante un diagrama de secuencia UML.

3.2.3. Autenticación interactiva mediante *sum-check*

El inicio de sesión interactivo comienza con `GET/api/zkp/login-challenge/:username`. El servidor recupera los datos del usuario y crea una instancia de `SumCheckVerifierSession`, cuyo estado inicial contiene el valor registrado, el valor final esperado y el punto $\mathbf{r}$. Esta instancia se guarda temporalmente en `loginSessions` y se identifica mediante un valor `loginId`.

El cliente reconstruye la tabla de evaluaciones utilizando la contraseña introducida y los pesos registrados. En cada ronda divide la tabla actual en dos mitades y calcula

$$L_j = g_j(0), \tag{3.6}$$

$$R_j = g_j(1). \tag{3.7}$$

Debido a la multilinealidad, el polinomio de ronda se representa como

$$g_j(t) = L_j + (R_j - L_j)t. \tag{3.8}$$

Los valores L_j y R_j se envían mediante `POST/api/zkp/login-round`. El servidor comprueba

$$L_j + R_j = C_j \tag{3.9}$$

y actualiza el valor mediante

$$C_{j+1} = (1 - r_j)L_j + r_j R_j. \tag{3.10}$$

El cliente realiza el mismo plegado sobre su tabla de evaluaciones, reduciendo su tamaño a la mitad en cada ronda.

En la implementación, el vector completo $\mathbf{r}$ se genera durante el registro, se almacena en PostgreSQL y se entrega al cliente al comenzar el login. Por tanto, existe interacción HTTP ronda a ronda, pero los retos no se generan de forma adaptativa después de cada mensaje como en la formulación canónica de sum-check. Esta decisión constituye una simplificación didáctica del prototipo, ya que como el cliente

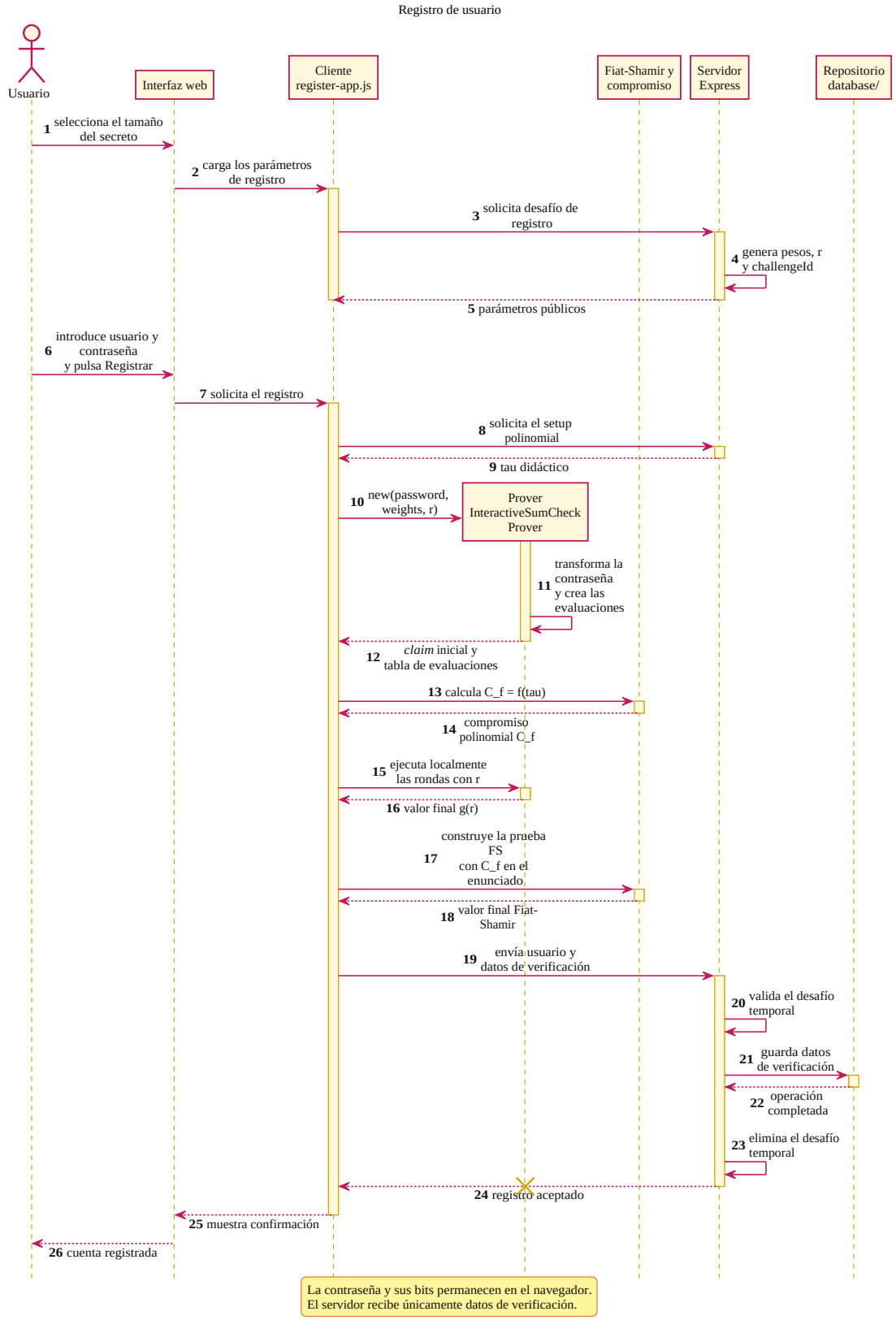

Figura 3.3: Diagrama de secuencia UML del proceso de registro.

conoce el punto de evaluación r desde el inicio, la aceptación se basa en que `final_value` se mantenga oculto en el servidor.

Después de completar las v rondas, el cliente envía el valor plegado mediante `POST/api/zkp/login-finish`. El servidor acepta la ejecución si

$$\texttt{foldedValue} = C_v = \texttt{final\_value}. \tag{3.11}$$

La primera igualdad comprueba la coherencia de las rondas y la segunda compara el resultado con el valor almacenado durante el enrolamiento.

La Figura 3.4 muestra la secuencia seguida durante la autenticación interactiva mediante un diagrama UML.

3.2.4. Autenticación mediante Fiat-Shamir

La modalidad no interactiva reutiliza la representación anterior, pero sustituye el punto almacenado **r** por retos derivados mediante SHA-256. El cliente obtiene los parámetros con `GET/api/zkp/login-challenge-fs/:username` y genera localmente todos los mensajes de ronda.

El enunciado incorporado al registro de ejecución contiene el separador de dominio `sumcheck-fs-v2`, el identificador del protocolo, el primo del cuerpo, el nombre de usuario, la longitud de la tabla, el número de rondas, el valor inicial, los pesos y el compromiso polinomial registrado. Para la ronda j, el reto se calcula como

$$r_j = H(\mathit{statement}, m_0, \ldots, m_j) \bmod p, \tag{3.12}$$

donde $m_j = (j, L_j, R_j)$. Si la reducción produce cero, la implementación utiliza el valor uno. Al incluir todos los mensajes anteriores, cada reto queda ligado al registro acumulado.

El cliente envía los mensajes y el valor final mediante `POST/api/zkp/login-fs`. No envía un nuevo valor inicial: el servidor utiliza el almacenado en PostgreSQL, reconstruye el enunciado, deriva nuevamente cada reto y comprueba el orden y la consistencia de todas las rondas. La prueba se acepta cuando

$$\texttt{foldedValue} = C_v = \texttt{final\_value\_fs}. \tag{3.13}$$

El compromiso polinomial forma parte del enunciado Fiat-Shamir siempre que exista en la cuenta. Por ello queda ligado al registro de interacción incluso cuando el usuario no activa la verificación de apertura. La opción de la interfaz controla únicamente si se realiza además dicha comprobación polinomial.

La Figura 3.5 resume en un diagrama de secuencia UML el flujo de autenticación mediante Fiat-Shamir.

3.2.5. Compromiso y apertura polinomial didácticos

La tabla $(g_0, \ldots, g_{N-1})$ se reutiliza como vector de coeficientes de un polinomio univariante:

$$f(X) = \sum_{i=0}^{N-1} g_i X^i. \tag{3.14}$$

Esta representación es distinta de la extensión multilineal utilizada por *sum-check* y se introduce únicamente para ilustrar las operaciones de compromiso, apertura y verificación.

El valor público τ se carga desde `trusted-setup.json` y se proporciona al navegador mediante `GET/api/poly-setup`. Durante el registro se calcula

$$C_f = f(\tau), \tag{3.15}$$

que se almacena en la columna `poly_commitment`.

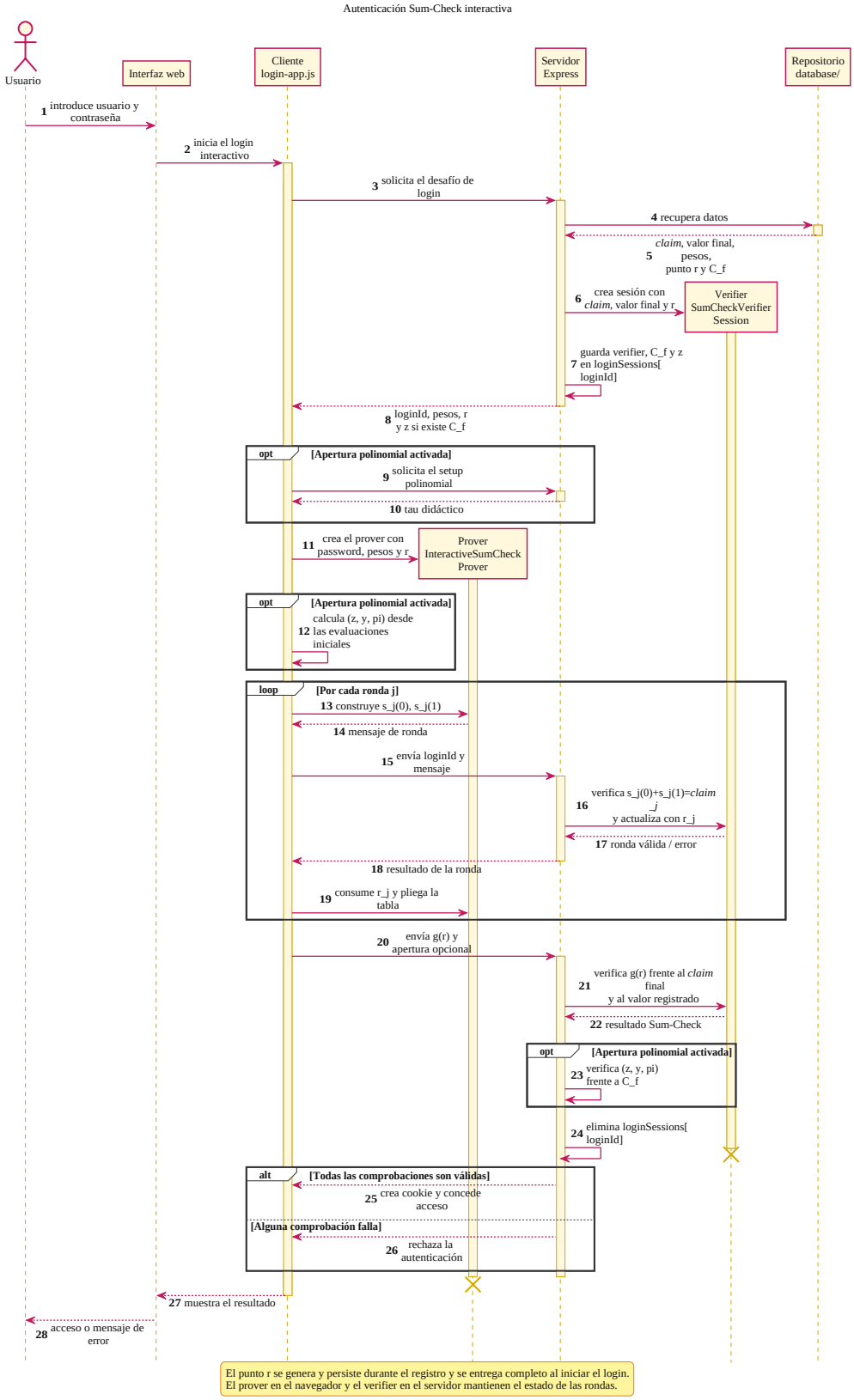

Figura 3.4: Diagrama de secuencia UML para la autenticación interactiva mediante *sum-check*. La generación y verificación de la apertura polinomial solo se ejecutan cuando esta opción está activada.

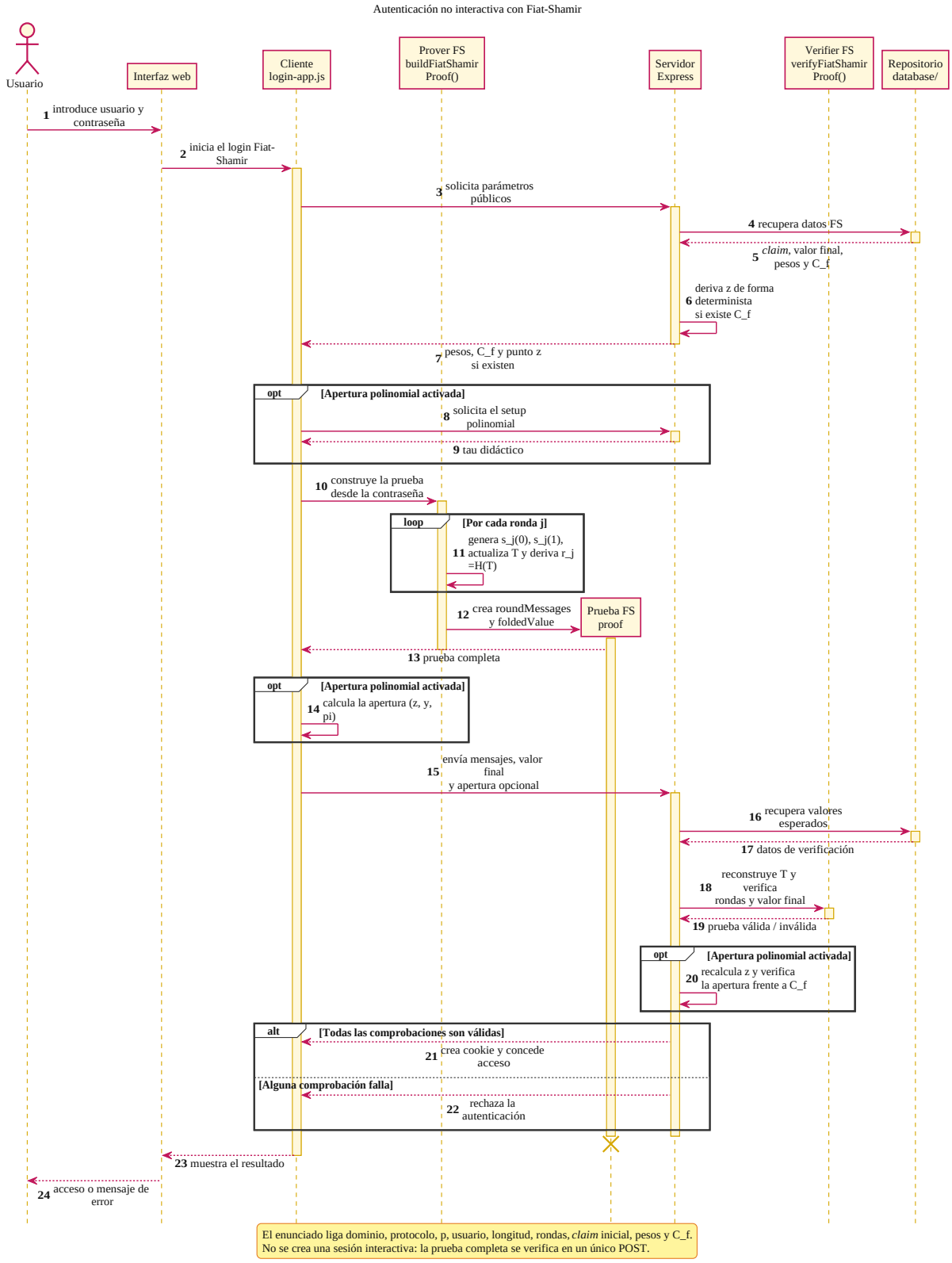

Figura 3.5: Diagrama de secuencia UML de la autenticación mediante Fiat-Shamir. La apertura polinomial constituye una comprobación adicional opcional.

Para abrir el polinomio en un punto z, el navegador calcula

$$y = f(z), \qquad q(X) = \frac{f(X) - y}{X - z}, \qquad \pi = q(\tau). \tag{3.16}$$

En el modo interactivo, z se genera aleatoriamente y queda almacenado en la sesión temporal. En Fiat-Shamir se deriva de forma determinista a partir del usuario, el valor inicial, el compromiso y la longitud, utilizando el separador de dominio `poly-opening-v1`. En ambos casos se evita que z coincida con τ.

Cuando la opción polinomial está activada, el servidor comprueba que el punto recibido coincide con el esperado y verifica

$$C_f - y = (\tau - z)\pi. \tag{3.17}$$

La autenticación solo se acepta si son válidas tanto las comprobaciones de *sum-check* como la apertura polinomial. Tras una autenticación satisfactoria, el servidor crea un token firmado mediante HMAC-SHA256 y lo almacena en una *cookie* HTTP para proteger el acceso a la zona privada.

La Figura 3.6 muestra en un diagrama de secuencia UML las operaciones de compromiso, apertura y verificación polinomial incorporadas al prototipo.

Esta capa no implementa KZG real: τ es público, no existe una SRS codificada en grupos criptográficos y no se emplean emparejamientos bilineales. Además, regenerar `trusted-setup.json` modifica τ e invalida los compromisos almacenados anteriormente.

El alcance de seguridad de los datos de registro también debe diferenciarse del de un sistema de autenticación resistente al compromiso del verificador. Los pesos públicos y el valor inicial forman una función determinista de la contraseña. Por ello, si se obtuvieran los datos almacenados, sería posible ensayar contraseñas candidatas fuera de línea, reconstruir su tabla y comparar los valores resultantes. El hecho de que la contraseña no se envíe en las peticiones evita su transmisión directa, pero no elimina este ataque sobre los datos de verificación [13].

Asimismo, la apertura polinomial es únicamente ilustrativa. Al ser τ público, un tercero que escogiera y podría calcular, para $z \neq \tau$,

$$\pi = \frac{C_f - y}{\tau - z} \tag{3.18}$$

y satisfacer la igualdad verificada sin conocer el polinomio original. La apertura reproduce por tanto la relación algebraica de una construcción tipo KZG, pero no añade una garantía criptográfica de autenticación. Estas limitaciones, junto con la confianza depositada en los valores remitidos por el cliente durante el enrolamiento y la generación anticipada de los retos interactivos, justifican la denominación de prototipo desarrollado como *ZK-like*.

3.3. Configuraciones implementadas

La interfaz permite seleccionar de forma independiente la transformación de Fiat-Shamir y la verificación de la apertura polinomial. De esta combinación se obtienen las cuatro configuraciones mostradas en la Tabla 3.1. El compromiso polinomial se genera durante el registro en todos los casos; la segunda opción determina únicamente si su apertura se comprueba durante la autenticación.

3.4. Despliegue y reproducibilidad

Con el objetivo de facilitar la reproducción del prototipo, se proporciona un despliegue contenerizado mediante Docker. La configuración divide el sistema en tres servicios. El servicio `db` ejecuta PostgreSQL 17 y conserva los datos de la base de datos en el volumen `postgres_data`. El servicio temporal `db-init` ejecuta `database/db_setup.js` para crear o actualizar el esquema. Finalmente, `app` contiene el servidor Node.js y Express, la interfaz web y los módulos del protocolo.

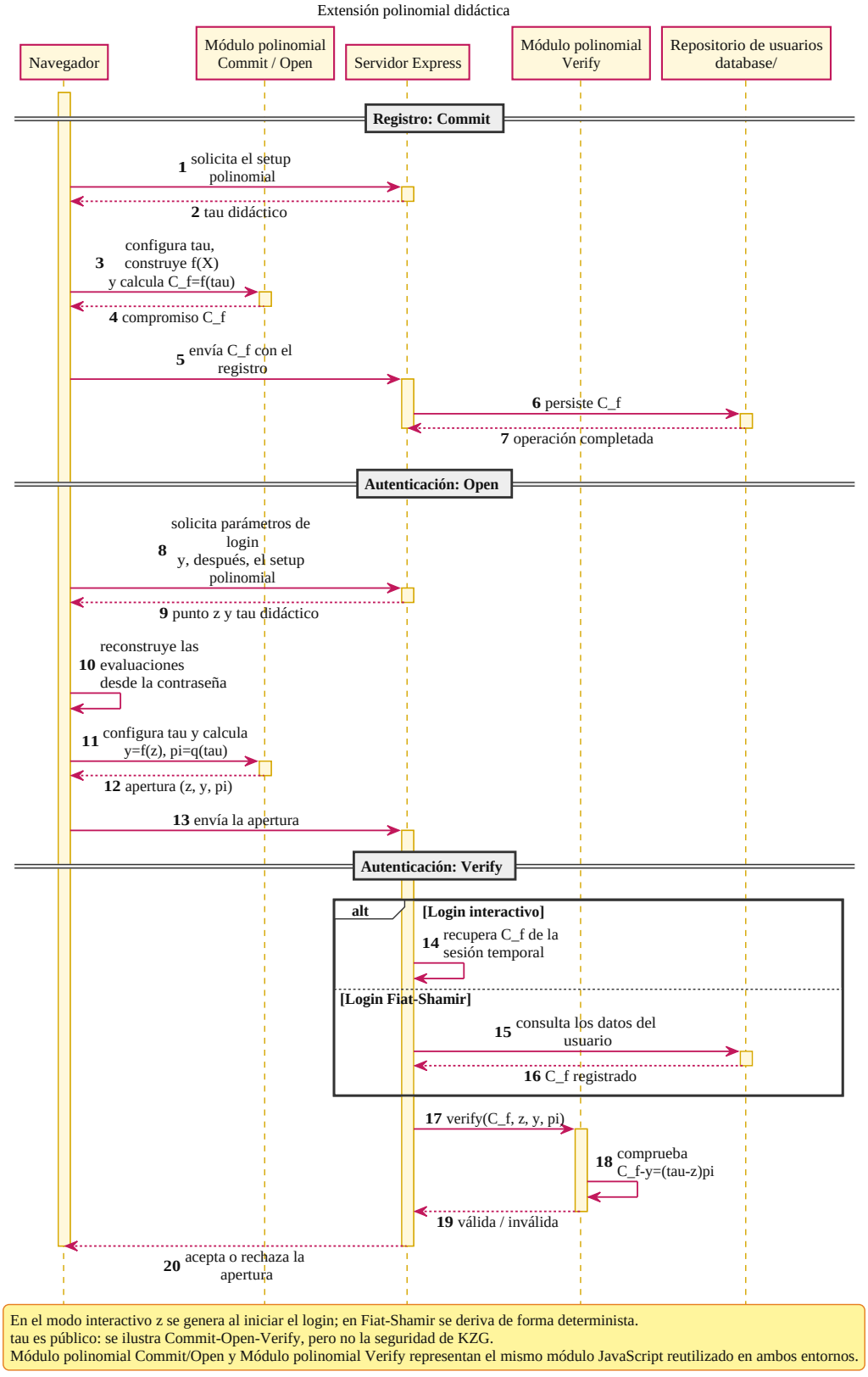

Figura 3.6: Diagrama de secuencia UML para las operaciones didácticas de compromiso, apertura y verificación polinomial.

Tabla 3.1: Configuraciones de autenticación implementadas.

Modalidad	**Interacción**	**Apertura**	**Comprobación principal**
sum-check interactivo	Ronda a ronda	No	Consistencia de las rondas y valor final registrado.
sum-check interactivo con apertura	Ronda a ronda	Sí	*sum-check* y verificación adicional de $C_f - y = (\tau - z)\pi$.
Fiat-Shamir	Una única petición	No	Reconstrucción de los retos y valor final Fiat-Shamir.
Fiat-Shamir con apertura	Una única petición	Sí	Fiat-Shamir y verificación adicional de la apertura.

Los contenedores se comunican a través de una red interna en la que PostgreSQL es accesible mediante el nombre `db`. Solo se publica el puerto HTTP de la aplicación, cuyo valor por defecto es el 3000, mientras que el puerto de la base de datos no se expone al equipo anfitrión. De este modo, el navegador accede al prototipo a través de `http://localhost:3000` y la persistencia permanece aislada dentro del entorno de Docker. La Figura 3.7 muestra la organización del despliegue.

Este despliegue está orientado a demostración y evaluación académica. La contenerización mejora la portabilidad y la reproducibilidad, pero no modifica las propiedades del protocolo. Un entorno de producción requeriría, entre otros aspectos, HTTPS, gestión externa de secretos, copias de seguridad, limitación de recursos y la sustitución de la capa polinomial didáctica por una construcción criptográfica adecuada.

El código desarrollado se encuentra en el siguiente repositorio de *GitHub*: `https://github.com/jorgeruuiz/zk-like-prototype`

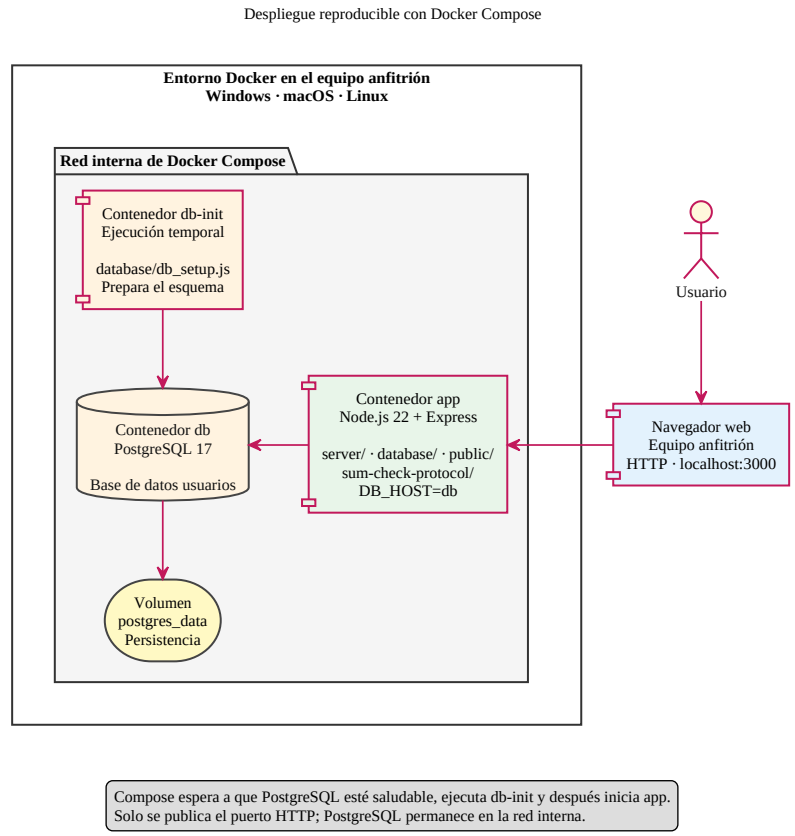

Figura 3.7: Despliegue reproducible del prototipo mediante Docker.

Capítulo 4

Evaluación

En este capítulo se evalúa el comportamiento del prototipo descrito en el capítulo anterior. En primer lugar, se presenta el entorno de ejecución y la metodología seguida para obtener medidas reproducibles. A continuación, se recogen las pruebas funcionales realizadas sobre los casos de aceptación y rechazo más representativos. Se comparan las cuatro configuraciones implementadas mediante las latencias observadas y se estudia si el navegador y el equipo anfitrión introducen diferencias apreciables en los resultados. Finalmente, se analiza el número de peticiones requerido por cada modalidad y se delimitan las condiciones bajo las que deben interpretarse las mediciones.

4.1. Entorno y metodología

La evaluación se realiza sobre el despliegue Docker descrito en la Sección 3.4. Con el fin de comprobar si el entorno de ejecución influye de forma apreciable en los resultados, las mediciones se repiten en dos equipos anfitriones y cuatro navegadores distintos, formando cinco combinaciones de navegador y entorno. En todos los casos se accede a la aplicación mediante `http://localhost:3000` y se utiliza el mismo flujo funcional del prototipo. La Tabla 4.1 resume los entornos de ejecución utilizados.

Tabla 4.1: Entornos utilizados en la evaluación de rendimiento.

Entorno	**Equipo anfitrión**	**Sistema Operativo**	**Navegador**
Safari/macOS	Apple M1, 8 núcleos, 16 GB RAM	macOS 26.5.1	Safari 26.5
Firefox/macOS	Apple M1, 8 núcleos, 16 GB RAM	macOS 26.5.1	Firefox 152.0.6
Edge/Windows	AMD Ryzen 7 5800HS, 16 procesadores lógicos, 16 GB RAM	Windows 11 Home 25H2 (26200.8737)	Edge 149.0.4022.98
Chrome/Windows	AMD Ryzen 7 5800HS, 16 procesadores lógicos, 16 GB RAM	Windows 11 Home 25H2 (26200.8737)	Chrome 148.0.7778.178
Firefox/Windows	AMD Ryzen 7 5800HS, 16 procesadores lógicos, 16 GB RAM	Windows 11 Home 25H2 (26200.8737)	Firefox 152.0.6

Dentro de los contenedores, la aplicación se ejecuta con Node.js 22.23.1 sobre Debian 12 y la base de datos con PostgreSQL 17.10. Respecto a Docker, se utiliza Docker Engine 29.5.3 y Docker Compose 5.1.4. Por tanto, las diferencias entre entornos se concentran principalmente en el equipo anfitrión, el sistema operativo, y el motor JavaScript del navegador. La construcción de las pruebas se realiza siempre con los módulos JavaScript cargados por el propio navegador, y las peticiones se envían mediante la API `fetch`.

Firefox 152.0.6 se ejecuta en ambos equipos con el objetivo de mantener constante el navegador en una parte de la comparación. Esto permite observar si el patrón relativo entre configuraciones se conserva al cambiar de entorno, aunque no aísla exclusivamente el efecto del sistema operativo ya que también difiere el hardware anfitrión.

La validación funcional se lleva a cabo manualmente sobre los controles de la interfaz. Las repeticiones necesarias para medir el rendimiento se realizan mediante el *benchmark* integrado en la pantalla de autenticación. Dicho *benchmark* ejecuta las cuatro configuraciones implementadas dentro de cada repetición antes de comenzar la siguiente. De este modo se limita el efecto de la evolución temporal del entorno, aunque las configuraciones se recorren siempre en el mismo orden.

Las mediciones se repiten para contraseñas ASCII de 4, 8, 16, 32, 64, 128 y 256 bytes. Estos tamaños producen tablas de 32 a 2048 evaluaciones y entre 5 y 11 rondas. El caso de 256 bytes se añade a modo de caso de estrés, ya que no corresponde con la longitud habitual de una contraseña.

Para cada tamaño se registra una cuenta y se activa la opción *Comparar rendimiento*. El navegador ejecuta cinco repeticiones de calentamiento y obtiene treinta muestras de cada configuración. Las trazas se desactivan durante la evaluación y el compromiso polinomial se carga durante el calentamiento, antes de conservar las mediciones finales.

La métrica principal es la latencia extremo a extremo calculada mediante `performance.now()`, que proporciona un reloj monótono apropiado para medir intervalos temporales en el navegador [14]. La medición comienza antes de solicitar el desafío y finaliza al recibir la respuesta del servidor. Por tanto, comprende la construcción de la prueba en el navegador, el intercambio HTTP local y el procesamiento de la respuesta. Para cada configuración se calculan la media, la mediana y el percentil 95 ($p95$). Los resultados se agrupan por tamaño de contraseña, configuración del protocolo y navegador utilizado.

4.2. Validación funcional

El objetivo de la validación funcional del sistema consiste en comprobar desde la interfaz que la implementación recorre las ramas descritas en el Capítulo 3 y que los errores de uso más directos son rechazados en el punto esperado. La Tabla 4.2 resume los casos ejecutados.

Tabla 4.2: Resultados de la validación funcional.

Caso	**Resultado esperado**	**Resultado obtenido**
Registro de una cuenta nueva	Persistencia de los datos de verificación.	Aceptado
Registro de un usuario duplicado	Rechazo por unicidad del nombre de usuario.	Rechazado (409)
Cuatro configuraciones con la contraseña correcta	Aceptación de sum-check o Fiat-Shamir y de la apertura cuando procede.	4 de 4 aceptadas
Contraseña incorrecta en modo interactivo	Fallo de la igualdad de consistencia de una ronda.	Rechazada (400)
Contraseña incorrecta en modo Fiat-Shamir	Fallo de la consistencia del registro reconstruido.	Rechazada (400)
Usuario inexistente	Ausencia de datos de verificación.	Rechazado (404)
Cookie, acceso privado y logout	Acceso tras autenticar y rechazo después del logout.	Comportamiento correcto

En el caso de la contraseña incorrecta, el servidor detectó que $g_j(0) + g_j(1)$ no coincidía con el valor de la primera ronda. En la modalidad Fiat-Shamir, la reconstrucción del registro produjo igualmente una prueba no válida. Las configuraciones con apertura polinomial se comprobaron tanto con la contraseña correcta como con una contraseña incorrecta, de modo que la apertura se mantuvo como una condición

adicional y no como un sustituto de sum-check o Fiat-Shamir. Una autenticación válida creó la *cookie* firmada y permitió solicitar `private.html`. Después de invocar `POST/api/auth/logout`, la consulta de la sesión devolvió un estado no autenticado.

4.3. Evaluación de rendimiento

Las abreviaturas I, I+A, FS y FS+A corresponden, respectivamente, al modo interactivo, interactivo con apertura polinomial, Fiat-Shamir y Fiat-Shamir con apertura polinomial. En las tablas siguientes se presenta la latencia extremo a extremo en milisegundos mediante el formato "media / $p95$".

4.3.1. Resultados por navegador

Las Tablas 4.3, 4.4, 4.5, 4.6 y 4.7 recogen los resultados obtenidos en las cinco combinaciones de navegador y entorno. En todos los casos se ha ejecutado el banco de pruebas integrado en la aplicación con la misma configuración.

Tabla 4.3: Latencia de autenticación en Safari: media / $p95$, en milisegundos.

Bytes	Rondas	I	I+A	FS	FS+A
4	5	20,33 / 25	20,80 / 25	9,07 / 10	10,40 / 13
8	6	23,77 / 28	25,83 / 31	10,63 / 12	12,53 / 14
16	7	27,90 / 30	31,80 / 34	13,97 / 16	17,80 / 20
32	8	35,50 / 39	43,07 / 46	19,93 / 22	26,87 / 30
64	9	46,83 / 51	59,80 / 63	29,57 / 32	43,30 / 47
128	10	67,37 / 72	90,27 / 93	48,70 / 52	74,33 / 78
256	11	151,27 / 432	237,60 / 736	144,73 / 435	234,27 / 696

Tabla 4.4: Latencia de autenticación en Firefox sobre macOS: media / $p95$, en milisegundos.

Bytes	Rondas	I	I+A	FS	FS+A
4	5	14,90 / 17	16,03 / 20	6,60 / 8	7,67 / 9
8	6	16,70 / 19	17,27 / 19	7,20 / 9	8,60 / 11
16	7	19,77 / 23	20,90 / 23	9,10 / 11	11,40 / 13
32	8	23,53 / 27	28,60 / 32	12,67 / 15	16,27 / 17
64	9	30,83 / 34	38,23 / 42	18,57 / 21	27,83 / 30
128	10	43,67 / 53	55,93 / 61	30,00 / 33	45,33 / 48
256	11	65,90 / 69	92,27 / 96	55,43 / 58	86,07 / 90

Tabla 4.5: Latencia de autenticación en Edge: media / $p95$, en milisegundos.

Bytes	Rondas	I	I+A	FS	FS+A
4	5	20,28 / 21	20,92 / 23	9,60 / 10	10,57 / 11
8	6	22,35 / 23	23,69 / 25	10,32 / 11	55,68 / 61
16	7	29,46 / 35	31,75 / 41	13,73 / 18	62,77 / 69
32	8	34,19 / 41	39,82 / 53	61,97 / 66	70,45 / 79
64	9	44,57 / 51	53,87 / 60	71,68 / 75	80,63 / 88
128	10	61,22 / 69	81,23 / 90	87,33 / 93	109,64 / 110
256	11	84,90 / 93	121,86 / 141	111,67 / 123	154,21 / 170

Tabla 4.6: Latencia de autenticación en Chrome: media / $p95$, en milisegundos.

Bytes	Rondas	I	I+A	FS	FS+A
4	5	24,39 / 26	24,75 / 26	10,85 / 12	11,39 / 11
8	6	26,18 / 29	27,30 / 30	11,55 / 12	57,07 / 65
16	7	31,03 / 34	33,97 / 39	13,85 / 16	62,22 / 65
32	8	37,34 / 43	41,14 / 44	63,17 / 72	70,37 / 74
64	9	49,27 / 63	59,00 / 83	72,15 / 96	80,63 / 89
128	10	58,22 / 60	72,94 / 76	79,35 / 83	100,25 / 102
256	11	96,55 / 120	132,35 / 155	114,88 / 133	157,23 / 179

Tabla 4.7: Latencia de autenticación en Firefox sobre Windows: media / $p95$, en milisegundos.

Bytes	Rondas	I	I+A	FS	FS+A
4	5	28,90 / 32	30,30 / 35	12,40 / 15	12,93 / 15
8	6	33,37 / 37	34,77 / 40	13,27 / 15	15,23 / 18
16	7	38,20 / 42	40,40 / 45	15,77 / 17	18,33 / 20
32	8	43,97 / 48	48,67 / 52	19,23 / 21	24,90 / 27
64	9	53,37 / 57	62,37 / 66	26,70 / 28	37,23 / 39
128	10	68,63 / 78	90,07 / 103	42,27 / 50	64,43 / 73
256	11	97,47 / 103	134,27 / 140	70,33 / 74	111,67 / 117

La Figura 4.1 representa las latencias medias de las cinco combinaciones de navegador y entorno con dimensiones y escala comunes. Entre 4 y 128 bytes se observa en Safari un crecimiento progresivo en las cuatro configuraciones. En ese intervalo, Fiat-Shamir fue más rápido que el modo interactivo. Para 8 bytes, la media descendió de 23,77 a 10,63 ms, una reducción del 55,3 %. Para 128 bytes pasó de 67,37 a 48,70 ms, una reducción del 27,7 %. La ventaja disminuye al crecer la tabla porque el coste de construir y verificar la prueba adquiere más peso que el ahorro de peticiones.

Firefox presenta un patrón similar en los dos equipos. Fiat-Shamir obtiene menor latencia que el modo interactivo para todos los tamaños evaluados, con y sin apertura polinomial. En macOS, para 8 bytes, la media pasa de 16,70 a 7,20 ms, y para 128 bytes de 43,67 a 30,00 ms. En Windows se reduce de 33,37 a 13,27 ms y de 68,63 a 42,27 ms, respectivamente. Por tanto, al mantener Firefox 152.0.6 en ambos

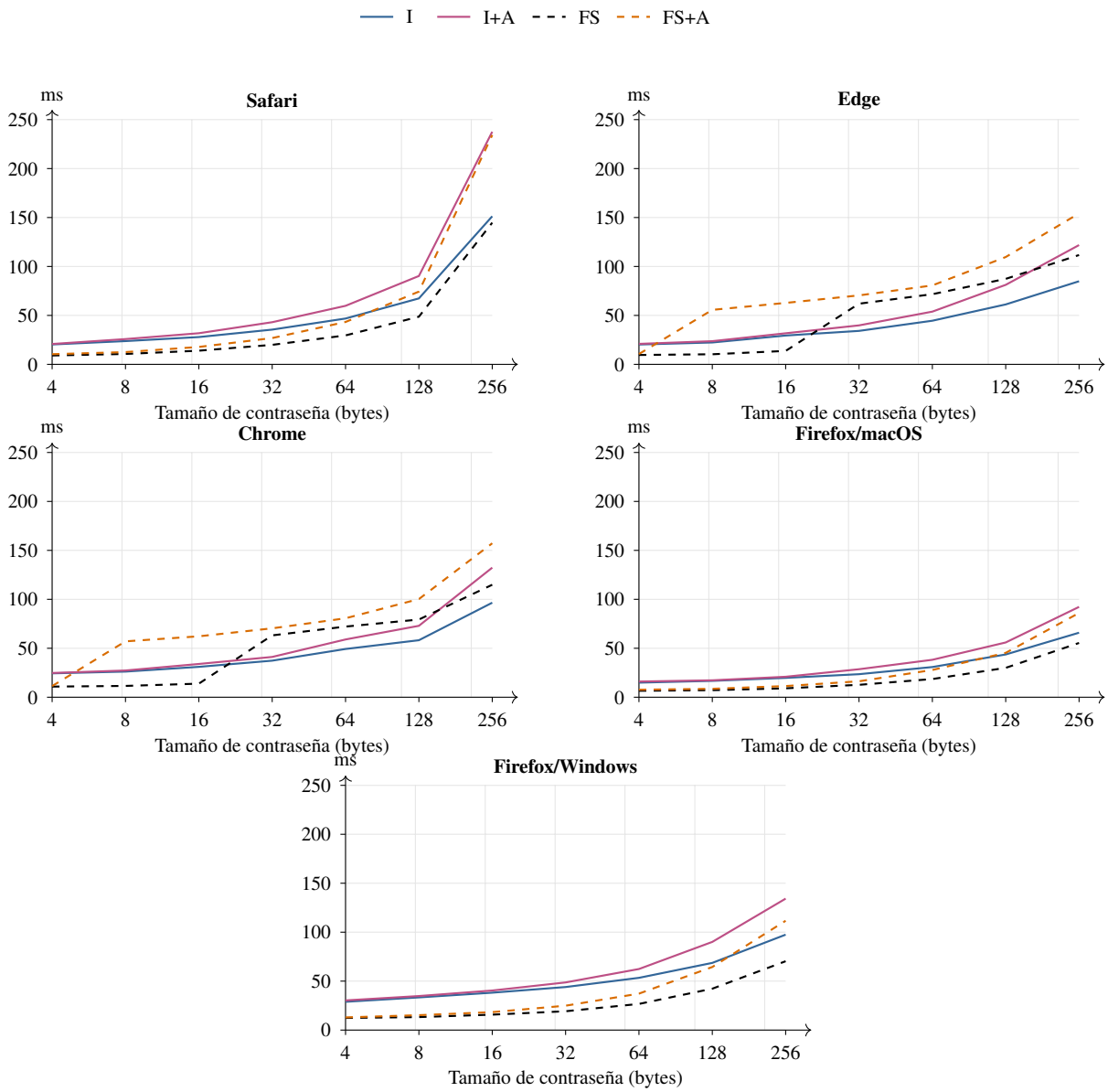

Figura 4.1: Latencia media de las cuatro configuraciones en las cinco combinaciones de navegador y equipo. Todos los paneles utilizan la misma escala.

entornos se conserva el mismo orden relativo entre configuraciones, aunque los tiempos absolutos son menores en macOS.

En Edge y Chrome aparece un patrón distinto. El modo Fiat-Shamir sin apertura es más rápido que el interactivo para 4, 8 y 16 bytes, pero a partir de 32 bytes pasa a tener una latencia mayor. En la variante con apertura, el incremento aparece ya desde 8 bytes. Esta diferencia no indica un fallo funcional: todas las ejecuciones fueron aceptadas y el servidor reconstruyó los mismos retos. El contenido del registro de ejecución es determinista; lo que puede variar entre navegadores es el coste de construirlo, serializarlo y procesar los *hashes* SHA-256.

La implementación de Fiat-Shamir concentra en una única ejecución la construcción completa de la prueba y deriva un reto por ronda a partir del registro acumulado. Este registro incluye el enunciado público con el vector de pesos, por lo que su coste de serialización crece con la tabla. La ausencia del incremento anterior en Firefox sobre el mismo equipo Windows es compatible con que el comportamiento observado en Edge y Chrome esté relacionado con su entorno de ejecución basado en Chromium. Sin embargo, las mediciones extremo a extremo no permiten atribuirlo a una operación interna concreta de su motor JavaScript o de WebCrypto.

La apertura polinomial añadió un coste creciente en los cinco entornos. Este comportamiento es coherente con la implementación, ya que se evalúan el polinomio y su cociente a partir de la tabla inicial. En la modalidad Fiat-Shamir con apertura, además, se construye primero la prueba no interactiva y después se reconstruye la tabla de evaluaciones para generar la apertura. El sobrecoste adquiere más peso a medida que aumenta la longitud de la entrada.

El caso de 256 bytes presentó en Safari una variabilidad considerablemente mayor. En el modo interactivo se obtuvo una mediana de 102,31 ms, frente a una media de 151,27 ms y un $p95$ de 432,00 ms. Con apertura, la mediana fue de 149,13 ms, la media de 237,60 ms y el $p95$ de 736,00 ms. El mismo fenómeno apareció en Fiat-Shamir. La distancia entre la mediana y la media indica la existencia de pausas ocasionales compatibles con factores como el recolector de basura, la compilación dinámica o la planificación del sistema. En Edge, Chrome y Firefox sobre ambos equipos, media, mediana y $p95$ permanecieron mucho más próximos. Aun así, 256 bytes representa una entrada alejada del uso habitual de una contraseña y se mantiene como prueba de estrés del prototipo.

4.3.2. Comparación entre navegadores y equipos

Para facilitar la comparación entre entornos, la Tabla 4.8 recoge la media agregada de las medias obtenidas entre 4 y 128 bytes. Se excluye el caso de 256 bytes porque se ha considerado una prueba de estrés y, en Safari, presenta una variabilidad considerablemente mayor.

Tabla 4.8: Resumen agregado de latencias medias en milisegundos entre 4 y 128 bytes.

Entorno	**I**	**I+A**	**FS**	**FS+A**
Safari/macOS	36,95	45,26	21,98	30,87
Firefox/macOS	24,90	29,49	14,02	19,52
Edge/Windows	35,35	41,88	42,44	64,96
Chrome/Windows	37,74	43,18	41,82	63,66
Firefox/Windows	44,41	51,10	21,61	28,84

Esta comparación no pretende aislar de forma exacta el coste de cada motor JavaScript, ya que también intervienen el sistema operativo, Docker, el hardware y la planificación del equipo anfitrión. La repetición con Firefox permite, no obstante, realizar dos comparaciones complementarias: el mismo navegador en equipos distintos y varios navegadores sobre un mismo equipo.

Al comparar Firefox 152.0.6 entre ambos entornos se conserva el mismo orden relativo de las configuraciones. Fiat-Shamir es más rápido que el modo interactivo y la apertura introduce un coste adicional en los dos casos. Sin embargo, macOS obtiene menores tiempos absolutos. En el resumen de 4 a 128 bytes, las medias de macOS son un 43,9 % menores en I, un 42,3 % en I+A, un 35,1 % en FS y un 32,3 % en FS+A respecto a Windows. Como también cambian el procesador y la plataforma de contenedores, estas diferencias deben atribuirse al entorno anfitrión en conjunto y no exclusivamente al sistema operativo.

En el equipo macOS, Firefox obtiene menores latencias agregadas que Safari en las cuatro configuraciones. Esta diferencia confirma que el navegador también puede influir en los tiempos absolutos aun cuando se mantiene el mismo equipo. No obstante, esta observación no permite establecer una clasificación general entre navegadores, pues se limita a un equipo y a las operaciones concretas de este prototipo.

La comparación dentro del equipo Windows aporta otra observación. Firefox es más lento que Edge y Chrome en las configuraciones interactivas agregadas, pero obtiene tiempos aproximadamente un 49 % menores en FS y un 55 % menores en FS+A. Por tanto, el resultado no puede explicarse como una ventaja general de Firefox: depende del tipo de trabajo ejecutado. Además, Firefox no reproduce los incrementos de Fiat-Shamir observados en Edge y Chrome a partir de determinados tamaños, pese a utilizar el mismo servidor y el mismo equipo Windows.

Edge y Chrome ofrecen resultados muy próximos entre sí, lo que es compatible con su base común Chromium. En conjunto, los datos sugieren que el coste adicional observado en sus modalidades Fiat-Shamir se concentra en el procesamiento realizado por el navegador. Esta interpretación no implica que el registro sea distinto: si el cliente y el servidor derivasen retos diferentes, la prueba sería rechazada. Para identificar la operación responsable sería necesario medir por separado la construcción de las tablas, la serialización del registro, cada llamada a SHA-256 y la verificación del servidor.

En consecuencia, las conclusiones deben distinguir entre propiedades estructurales del protocolo y resultados dependientes del entorno. El número de peticiones de cada modalidad es invariable, y el crecimiento del coste de la apertura aparece en todos los casos. En cambio, la latencia absoluta y la ventaja temporal de Fiat-Shamir dependen del navegador y del equipo utilizados.

4.3.3. Número de peticiones

Además de la medición temporal, el número de peticiones puede deducirse directamente de los flujos implementados. En el modo interactivo se realiza una petición para obtener el desafío, una por cada ronda y una última para finalizar el protocolo. Por tanto, para r rondas se requieren $r+2$ peticiones. Fiat-Shamir utiliza siempre dos: una para recuperar los parámetros y otra para enviar la prueba completa.

La apertura polinomial no modifica este recuento, ya que sus datos se incluyen en la petición final de cualquiera de las dos modalidades. Este análisis no constituye una medición de tráfico de red, sino una consecuencia de la secuencia de *endpoints* descrita en el Capítulo 3. Por ello, el número de peticiones no depende del navegador ni del equipo utilizado.

4.4. Limitaciones de la evaluación

Los resultados permiten comparar las configuraciones dentro del prototipo, pero no deben interpretarse como medidas generales de *sum-check*, Fiat-Shamir o los compromisos KZG. Aunque las pruebas se han repetido en dos equipos, cuatro navegadores y cinco combinaciones de ejecución, todas las mediciones se han realizado en local, sobre un despliegue Docker ejecutado en el propio equipo anfitrión. No se ha estudiado el efecto de una red real, de la pérdida de paquetes, de la latencia entre cliente y servidor ni del acceso concurrente de varios usuarios.

Las mediciones se obtuvieron mediante `performance.now()` dentro del navegador. Por tanto, incluyen la ejecución JavaScript, la construcción de las pruebas, las peticiones HTTP locales y el procesamiento de

la respuesta, pero no permiten separar con precisión el coste de cada operación interna. Tampoco incluyen el tiempo de interacción humana previo a pulsar el botón de autenticación o *benchmark*.

Los tiempos observados son pequeños y pueden verse afectados por factores externos al protocolo, como la planificación del sistema operativo, la compilación dinámica del motor JavaScript, el recolector de basura, Docker o procesos en segundo plano del equipo anfitrión. El uso de calentamientos, treinta repeticiones y el percentil 95 reduce el impacto de ejecuciones anómalas, pero no lo elimina por completo. Por este motivo, las diferencias pequeñas entre navegadores deben interpretarse con cautela.

La ejecución de Firefox 152.0.6 en ambos equipos reduce la variación debida al navegador, pero no constituye un experimento que aísle el sistema operativo. Los equipos utilizan procesadores distintos y cada plataforma integra Docker de forma diferente. Del mismo modo, la comparación entre Firefox, Edge y Chrome en Windows permite mantener constante el servidor anfitrión, pero no separa dentro del navegador el coste del motor JavaScript, WebCrypto, la serialización y la planificación de tareas. Las explicaciones propuestas para los incrementos de Fiat-Shamir en Chromium deben entenderse por tanto como inferencias compatibles con los datos, no como una identificación causal de una operación concreta.

Las contraseñas empleadas en rendimiento fueron cadenas ASCII sintéticas de longitud fija. Se estudió el efecto del tamaño de entrada, pero no el de distintas distribuciones de bits, alfabetos, codificaciones o patrones de contraseña. Además, solo se midió el proceso de autenticación. El registro se ejecuta una vez por cuenta y genera los datos de verificación de las dos modalidades, por lo que su coste no es directamente comparable con el de un intento de login.

Por último, la apertura polinomial utiliza un τ público y aritmética en el cuerpo finito, sin grupos criptográficos ni emparejamientos bilineales. Su rendimiento no representa el de una implementación KZG real. Del mismo modo, la validación funcional acredita el comportamiento observado para los casos ejecutados, pero no sustituye un análisis formal de seguridad ni una auditoría criptográfica. La evaluación debe entenderse como una comprobación experimental de un prototipo didáctico y de las decisiones de arquitectura adoptadas.

Capítulo 5

Conclusiones y trabajo futuro

En este trabajo se ha diseñado e implementado un prototipo académico de autenticación basado en técnicas algebraicas relacionadas con las pruebas de conocimiento cero, permitiendo transformar contraseñas de usuarios en construcciones matemáticas.

El prototipo desarrollado permite ejecutar un flujo completo de registro y autenticación en una arquitectura cliente-servidor. La contraseña se procesa en el navegador y no se envía directamente al servidor. En su lugar, se generan valores algebraicos que permiten comprobar autenticaciones posteriores. Sobre esta base se han implementado dos modalidades: una versión interactiva simplificada de *sum-check* y una versión no interactiva obtenida mediante Fiat-Shamir. También se ha incorporado una capa experimental de compromiso polinomial para representar de forma sencilla las operaciones de compromiso, apertura y verificación, y para poder conectar el prototipo con algunas ideas presentes en construcciones zk-SNARK.

La evaluación realizada confirma el comportamiento esperado de las configuraciones implementadas desde el punto de vista funcional. La modalidad no interactiva reduce el número de peticiones respecto al modo interactivo, aunque su impacto temporal depende del entorno de ejecución. Fiat-Shamir obtiene menores latencias en Safari y en Firefox sobre macOS y Windows. En Edge y Chrome se observa en cambio un coste mayor desde 32 bytes en la variante sin apertura y ya desde 8 bytes cuando se incluye la apertura polinomial. Firefox 152.0.6 conserva el mismo orden relativo entre configuraciones en ambos equipos, pero obtiene menores tiempos absolutos en macOS. Además, Firefox sobre Windows no reproduce los incrementos observados en los navegadores Chromium del mismo equipo, lo que sugiere que la diferencia está relacionada con el procesamiento realizado por el navegador y no con el flujo lógico del protocolo. La apertura polinomial introduce en todos los entornos un sobrecoste que crece con la tabla. El tamaño de 256 bytes se considera una prueba de estrés; su variabilidad fue especialmente elevada en Safari, pero no en el resto de entornos.

El trabajo también pone de manifiesto varias limitaciones. El sistema no proporciona conocimiento cero formal, no implementa un compromiso KZG real y simplifica algunos aspectos del protocolo *sum-check* interactivo, especialmente en la generación de retos. Además, los datos de verificación permiten comprobar contraseñas candidatas fuera de línea si se compromete la base de datos, y la apertura polinomial con τ público no aporta vinculación criptográfica. Por tanto, el resultado no debe interpretarse como una solución criptográfica lista para producción, sino como un prototipo académico orientado a comprender mecanismos algebraicos que forman parte de algunos sistemas modernos de pruebas.

Como trabajo futuro, una primera línea de continuación sería sustituir la capa polinomial didáctica por un esquema de compromiso polinomial real. En particular, podría estudiarse la integración de un compromiso KZG basado en grupos criptográficos, emparejamientos bilineales y una SRS generada correctamente. Esto permitiría acercar el prototipo a construcciones zk-SNARK reales.

Otra mejora consistiría en ajustar la modalidad interactiva a la formulación canónica de *sum-check*. En la implementación actual, el vector de retos se genera durante el registro y se reutiliza posteriormente. De forma alternativa, el servidor podría generar cada reto de forma adaptativa después de recibir el mensaje

correspondiente del *prover*, reproduciendo con mayor fidelidad el protocolo teórico.

Los resultados de rendimiento también motivan una línea de trabajo centrada en el perfilado y la optimización de Fiat-Shamir. Podrían medirse por separado la construcción de la tabla, la serialización del registro de interacción, las llamadas a SHA-256 y la verificación del servidor. A partir de ese análisis sería posible evitar serializaciones repetidas del enunciado público y reutilizar la tabla inicial al generar la apertura polinomial. Cualquier optimización debería mantener inalterada la definición canónica del registro y comprobar que cliente y servidor continúan derivando los mismos retos.

Por último, en este trabajo se ha utilizado una representación sencilla de la contraseña como tabla de evaluaciones. Una continuación natural sería expresar afirmaciones más ricas mediante circuitos aritméticos o sistemas de restricciones, acercando el prototipo a la estructura utilizada en zk-SNARKs.

Anexos

Anexo A

Planificación y dedicación

Este anexo recoge una estimación de la planificación temporal y de la dedicación asociada al desarrollo del Trabajo Fin de Grado hasta el 23 de julio de 2026. La planificación no debe interpretarse como una división estrictamente secuencial, ya que varias tareas se desarrollaron de forma solapada. En particular, el estudio teórico, la implementación del prototipo, la revisión de la memoria y la evaluación se fueron realimentando durante el desarrollo. La línea dedicada a la memoria agrupa tanto las 30 horas contabilizadas inicialmente como las 35 horas posteriores destinadas a las nuevas mediciones, la actualización del documento y la preparación de la presentación.

La Tabla A.1 resume la dedicación aproximada por línea de trabajo. La estimación combina el trabajo de estudio, implementación, pruebas, documentación y revisión realizado durante el proyecto.

La Figura A.1 muestra una representación temporal de estas líneas de trabajo. El diagrama refleja los solapes entre tareas: el estudio teórico acompaña a la implementación durante buena parte del proyecto, la redacción comienza antes de finalizar completamente el prototipo y la evaluación se desarrolla en paralelo con los últimos ajustes de despliegue y documentación.

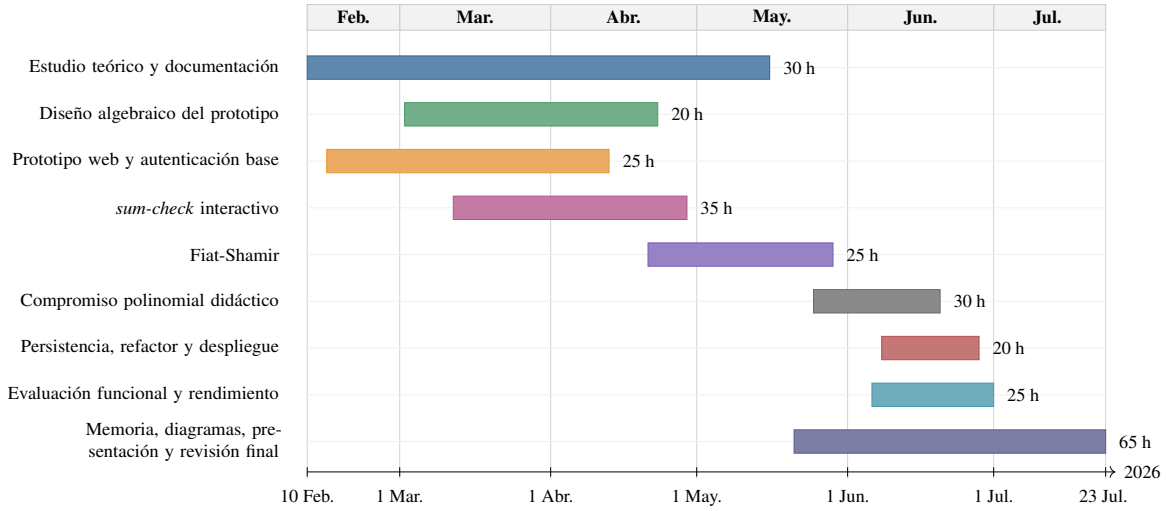

Figura A.1: Diagrama de Gantt por líneas de trabajo, con solapes entre estudio, implementación, evaluación, redacción y preparación de la presentación.

Tabla A.1: Dedicación estimada por línea de trabajo.

Línea de trabajo	**Descripción**	**Horas**
Estudio teórico y documentación	Revisión de pruebas interactivas, conocimiento cero, *sum-check*, Fiat-Shamir, compromisos polinomiales y relación con zk-SNARKs.	30
Diseño algebraico del prototipo	Definición de la representación binaria de la contraseña, selección del cuerpo finito, construcción de la tabla de evaluaciones y formulación del valor inicial.	20
Prototipo web y autenticación base	Desarrollo de la aplicación cliente-servidor, formularios de registro y login, comunicación HTTP/JSON, gestión de sesión y zona privada.	25
Implementación de *sum-check* interactivo	Implementación de los módulos de *prover* y *verifier*, generación de rondas, plegado de evaluaciones y comprobación del valor final.	35
Transformación de Fiat-Shamir	Construcción de la modalidad no interactiva, generación de retos mediante *hash* del registro y verificación completa de la prueba en el servidor.	25
Compromiso polinomial didáctico	Implementación de compromiso, apertura y verificación polinomial, junto con el *trusted setup* simulado y su integración opcional en las autenticaciones.	30
Persistencia, refactor y despliegue	Integración con PostgreSQL, separación de repositorio de usuarios, configuración de Docker Compose, scripts de ejecución y comprobaciones de salud.	20
Evaluación funcional y rendimiento	Pruebas de aceptación y rechazo, *benchmark* integrado, mediciones por tamaños de contraseña, comparación de configuraciones y análisis de resultados.	25
Memoria, diagramas, presentación y revisión final	Elaboración de la memoria y los diagramas, incorporación de las mediciones finales con Firefox, revisión de resultados y conclusiones, preparación de anexos y desarrollo inicial de la presentación.	65
Total	**Dedicación estimada del Trabajo Fin de Grado.**	**275**

Bibliografía

[1] S. Goldwasser, S. Micali, and C. Rackoff. *The Knowledge Complexity of Interactive Proof Systems*. SIAM Journal on Computing, vol. 18, no. 1, pp. 186–208, 1989.

[2] A. Fiat and A. Shamir. *How to Prove Yourself: Practical Solutions to Identification and Signature Problems*. Advances in Cryptology—CRYPTO '86, LNCS 263, pp. 186–194, 1987.

[3] D. Pointcheval and J. Stern. *Security Proofs for Signature Schemes*. Advances in Cryptology—EUROCRYPT '96, LNCS 1070, pp. 387–398, 1996.

[4] C. Lund, L. Fortnow, H. Karloff, and N. Nisan. *Algebraic Methods for Interactive Proof Systems*. Journal of the ACM, vol. 39, no. 4, pp. 859–868, 1992.

[5] A. Kate, G. M. Zaverucha, and I. Goldberg. *Constant-Size Commitments to Polynomials and Their Applications*. Advances in Cryptology—ASIACRYPT 2010, LNCS 6477, pp. 177–194, 2010.

[6] J. Groth. *On the Size of Pairing-Based Non-interactive Arguments*. Advances in Cryptology—EUROCRYPT 2016, LNCS 9666, pp. 305–326, 2016.

[7] A. Chiesa, M. A. Forbes, and N. Spooner. *A Zero Knowledge Sumcheck and its Applications*. Cryptology ePrint Archive, Report 2017/305, 2017.

[8] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev. *Scalable, Transparent, and Post-Quantum Secure Computational Integrity*. Cryptology ePrint Archive, Report 2018/046, 2018.

[9] B. Bünz, J. Bootle, D. Boneh, A. Poelstra, P. Wuille, and G. Maxwell. *Bulletproofs: Short Proofs for Confidential Transactions and More*. IEEE Symposium on Security and Privacy, pp. 315–334, 2018.

[10] J. Thaler. *Proofs, Arguments, and Zero-Knowledge*. Versión de 18 de julio de 2023. `https://people.cs.georgetown.edu/jthaler/ProofsArgsAndZK.pdf`. Accedido el 4 de septiembre de 2026.

[11] A. Nitulescu. *zk-SNARKs: A Gentle Introduction*. École Normale Supérieure, 2020. `https://www.di.ens.fr/~nitulesc/files/Survey-SNARKs.pdf`. Accedido el 4 de septiembre de 2026.

[12] R. Lavin, X. Liu, H. Mohanty, L. Norman, G. Zaarour, and B. Krishnamachari. *A Survey on the Applications of Zero-Knowledge Proofs*. arXiv:2408.00243 [cs.CR], 2024.

[13] National Institute of Standards and Technology. *Digital Identity Guidelines: Authentication and Authenticator Management*. NIST Special Publication 800-63B-4, July 2025. Accedido el 4 de septiembre de 2026.

[14] World Wide Web Consortium. *High Resolution Time, Level 3*. W3C Working Draft, March 24, 2026. `https://www.w3.org/TR/hr-time-3/`. Accedido el 4 de septiembre de 2026.