TRABAJO FIN DE GRADO

INGENIERÍA INFORMÁTICA

Ataques clásicos y cuánticos al RSA

Estudio del criptosistema RSA y sus potenciales debilidades ante las nuevas tecnologías clásicas y cuánticas

Autor

Jorge López Remacho

Directores

Jesús García Miranda

Antón Rodríguez Otero

Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación

—

Granada, junio de 2024

Ataques clásicos y cuánticos al RSA

Estudio del criptosistema RSA y sus potenciales debilidades ante las nuevas tecnologías clásicas y cuánticas

Autor
Jorge López Remacho

Directores
Jesús García Miranda
Antón Rodríguez Otero

Ataques clásicos y cuánticos al RSA

Jorge López Remacho

Palabras clave: criptografía, RSA, computación cuántica, Shor, factorización, seguridad, post-cuántica.

Resumen

En 1977, tres investigadores describieron el RSA, un sistema criptográfico revolucionario. Hoy en día, casi medio siglo después, esta tecnología sigue siendo fundamental en la protección de datos. La seguridad de numerosos sistemas, como protocolos de red HTTPS y SSL/TLS, se basa en su fortaleza, ya que las claves necesarias para el intercambio de información segura se transmiten normalmente cifradas con RSA. Sin embargo, el desarrollo de la computación cuántica amenaza con socavar la seguridad de este criptosistema.

En este trabajo, realizamos un estudio exhaustivo del RSA, examinando sus debilidades y los posibles ataques que puede enfrentar. Analizamos cómo la computación cuántica afecta su seguridad, explorando los requisitos necesarios de los ordenadores cuánticos para llevar a cabo estos ataques. Profundizaremos en el algoritmo de Shor y su capacidad para factorizar números grandes, lo que representa una amenaza directa para RSA.

Finalmente, investigamos nuevas formas de criptografía post-cuántica. Nos adentramos brevemente, mediante simulaciones, en cómo realizar intercambios de claves de forma segura a través de canales cuánticos. Este estudio nos permitirá entender mejor las alternativas que podrían reemplazar a RSA en un futuro donde la computación cuántica sea predominante, asegurando así la continuidad de la confidencialidad y seguridad en las comunicaciones digitales.

Classical and Quantum Attacks on RSA

Jorge López Remacho

Keywords: Criptography, RSA, quantum computing, Shor, factorization, security, post-quantum

Abstract

In 1977, three researchers described RSA, a revolutionary cryptographic system. Today, nearly half a century later, this technology remains fundamental in data protection. The security of numerous systems, such as HTTPS and SSL/TLS network protocols, relies on its strength, as the keys necessary for secure information exchange are usually encrypted with RSA. However, the development of quantum computing threatens to undermine the security of this cryptosystem.

In this work, we conduct an in-depth study of RSA, examining its weaknesses and potential attacks. We analyze how quantum computing affects its security, exploring the necessary requirements for quantum computers to carry out these attacks. We delve into Shor's algorithm and its ability to factor large numbers, which poses a direct threat to RSA.

Finally, we investigate new forms of post-quantum cryptography. Through simulations, we briefly explore how to perform secure key exchanges via quantum channels. This study will help us better understand the alternatives that could replace RSA in a future where quantum computing is predominant, thus ensuring the continuity of confidentiality and security in digital communications.

Yo, **Jorge López Remacho**, alumno de la titulación **Ingeniería Informática** de la **Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI 26597658K, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Jorge López Remacho

Granada a 24 de junio de 2024.

D. **Jesús García Miranda**, Profesor del Departamento de Álgebra de la Universidad de Granada.

D. **Antón Rodríguez Otero**, Quantum Computing Scientist en Fujitsu Technology Solutions.

Informan:

Que el presente trabajo, titulado ***Ataques clásicos y cuánticos al RSA***, ha sido realizado bajo su supervisión por **Jorge López Remacho**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a 25 de junio de 2024.

Los directores:

Jesús García Miranda **Antón Rodríguez Otero**

Agradecimientos

Tal y como una persona es ella y sus circunstancias, este trabajo no está completo sin hacer referencia a las personas a mi alrededor que lo han hecho posible.

Primero, quisiera expresar mi profunda gratitud a mis tutores, Jesús y Antón. Gracias por acompañarme durante estos cuatro meses de intenso trabajo, por confiar en mí y en un proyecto que se aleja de los conocimientos tradicionales adquiridos en la carrera. Vuestra guía, consejos, apoyo y amabilidad han sido invaluables.

A mis compañeros y amigos que he tenido el placer de conocer durante la carrera, con quienes he compartido innumerables momentos y luchado contra muchas asignaturas, tampoco puedo dejarlos fuera. Gracias por vuestra compañía, jovialidad y constante apoyo.

Quiero también agradecer a mi jefe, Pablo Romero, por sus valiosos consejos y por permitirme una semana de vacaciones para terminar este Trabajo Fin de Grado.

Finalmente, un agradecimiento muy especial a mis padres, por proporcionarme el sustento que me ha permitido llegar hasta la presentación de este trabajo y la obtención del Grado. Gracias por el cariño y la confianza que me transmitís a diario.

Índice general

Parte I

Introducción

Capítulo 1

Introducción

1.1. Contexto histórico y motivación

Desde tiempos inmemoriales, el ser humano ha necesitado comunicarse de forma confidencial. Cuando dicha comunicación es necesaria entre dos interlocutores alejados, surgen multitud de problemas: ¿cómo podemos asegurarnos de que el mensaje no sea leído por el mensajero o cualquier tercero antes de llegar a manos del receptor?

En la Antigua Grecia, alrededor del siglo V a.C., los espartanos ya eran conscientes de este problema. Una de las primeras soluciones que diseñaron fue la *escítala lacedemonia*: una vara en la que se enrollaba un pergamino con letras escritas en vertical. Si dicho pergamino se leía de forma lineal, el mensaje consistía en una secuencia de letras griegas sin sentido, mientras que si se colocaba alrededor de la escítala, las letras se ordenaban correctamente y el mensaje se hacía legible. Este fue uno de los primeros sistemas criptográficos conocidos.

Con el paso del tiempo, las técnicas criptográficas evolucionaron significativamente. Durante la Edad Media, se utilizaron métodos como el cifrado por sustitución y el cifrado por transposición. En el Renacimiento, figuras como Leon Battista Alberti y Blaise de Vigenère contribuyeron al desarrollo de sistemas más complejos, como el cifrado polialfabético. En la Segunda Guerra Mundial, la criptografía jugó un papel crucial con la creación y posterior descifrado de la máquina Enigma por parte de los aliados, un logro que cambió el curso de la historia.

Hoy en día, enfrentamos el mismo problema de confidencialidad que nuestros antepasados, salvo que hemos sustituido los mensajeros de carne y hueso por cables y antenas, y las diferentes letras del alfabeto por ristras de ceros y unos. Para proteger la información privada que viaja a través de internet, desde transacciones bancarias hasta conversaciones personales, en las

últimas décadas se han desarrollado multitud de sistemas criptográficos, los cuales van siendo sustituidos por sistemas cada vez más sofisticados conforme se descubren vulnerabilidades en los anteriores.

En 1977, tres investigadores del Instituto Tecnológico de Massachusetts (Ron Rivest, Adi Shamir y Leonard Adleman) describieron el algoritmo RSA, un sistema criptográfico que permite tanto cifrar y descifrar mensajes como autenticar al emisor del mismo. Este sistema, basado en la dificultad de factorizar grandes números primos, revolucionó el campo de la criptografía y se convirtió en uno de los pilares fundamentales de la seguridad informática.

Actualmente, casi medio siglo después, RSA sigue vigente a pesar de los múltiples intentos de vulnerarlo y de diseñar ataques contra él por parte de muchos criptoanalistas. La seguridad de muchos sistemas, como los protocolos de red HTTPS y SSL/TLS, o buena parte de los sistemas de firma digital actuales, se apoya en su fortaleza. Las claves necesarias para el intercambio de información segura se transmiten normalmente cifradas con RSA.

Sin embargo, el desarrollo de la computación cuántica está abriendo un gran abanico de posibilidades en el ámbito de la informática, planteando nuevos ataques sobre RSA. Algoritmos cuánticos, como el de Shor, prometen factorizar números grandes con una eficiencia sin precedentes, lo que amenaza con hacer obsoletos muchos de los sistemas criptográficos actuales. En este contexto, es crucial evaluar la resiliencia de RSA frente a estas nuevas amenazas y explorar alternativas viables que garanticen la seguridad de la información en un futuro dominado por la computación cuántica.

En esta línea, muchos investigadores han estudiado las nuevas posibilidades del paradigma cuántico para crear nuevas formas de criptografía. Una de ellas, que en un futuro post-cuántico podría reemplazar a RSA, son los protocolos de intercambio de claves por canales cuánticos. Estos protocolos hacen uso de las propiedades de la mecánica cuántica para garantizar un intercambio de claves teóricamente invulnerable.

1.2. Objetivos

En este Trabajo Fin de Grado, nos hemos propuesto los siguientes objetivos:

1. Realizar un estudio profundo del criptosistema RSA, haciendo hincapié en sus fundamentos matemáticos, debilidades y posibles ataques.

2. Evaluar la seguridad del criptosistema de forma teórica desde el punto de vista de la computación clásica y cuántica.

3. Implementar y analizar diferentes métodos de factorización clásicos co-

mo el método de Fermat, Pollard y la criba cuadrática para determinar su eficiencia.

4. Estudiar e implementar ataques diseñados específicamente para RSA, como el ataque de Wiener.

5. Implementar y analizar en profundidad el algoritmo de Shor, poniendo en valor su impacto actual en la seguridad del criptosistema RSA.

6. Hacer uso de simuladores y ordenadores cuánticos reales para analizar los resultados de la implementación y valorar el potencial actual de esta futura tecnología.

7. Estudiar protocolos de intercambio de claves por canales cuánticos, analizando su funcionamiento e implementando una prueba de concepto de los mismos en simuladores cuánticos.

1.3. Planificación

Para cumplir con los anteriores objetivos, proponemos la planificación expuesta en el diagrama de Gannt de la figura 1.1 para desarrollar este trabajo.

Dado que nos adentramos en dos ámbitos de conocimiento nuevos, los primeros días los destinaremos a estudiar tanto criptografía como computación cuántica. Así, consolidaremos una base lo suficientemente sólida como para poder afrontar este trabajo.

Posteriormente, redactaremos en la memeria una síntesis de dichos conocimientos básicos. También aprovecharemos para realizar buena parte de la implementación del Algoritmo de Shor, el cual tendremos recién estudiado de la etapa anterior.

Después de introducir los conocimientos básicos, comenzamos a adentrarnos en la redacción de la memoria sobre ataques clásicos y cuánticos, a la vez que seguimos aprendiendo del tema por esta vía.

Finalmente, una vez hayamos cubierto todo el tema teórico, pasaremos a la implementación técnica de los ataques clásicos y los protocolos de intercambio de claves. La última semana la dedicaremos a redactar y sacar las conclusiones del trabajo realizado.

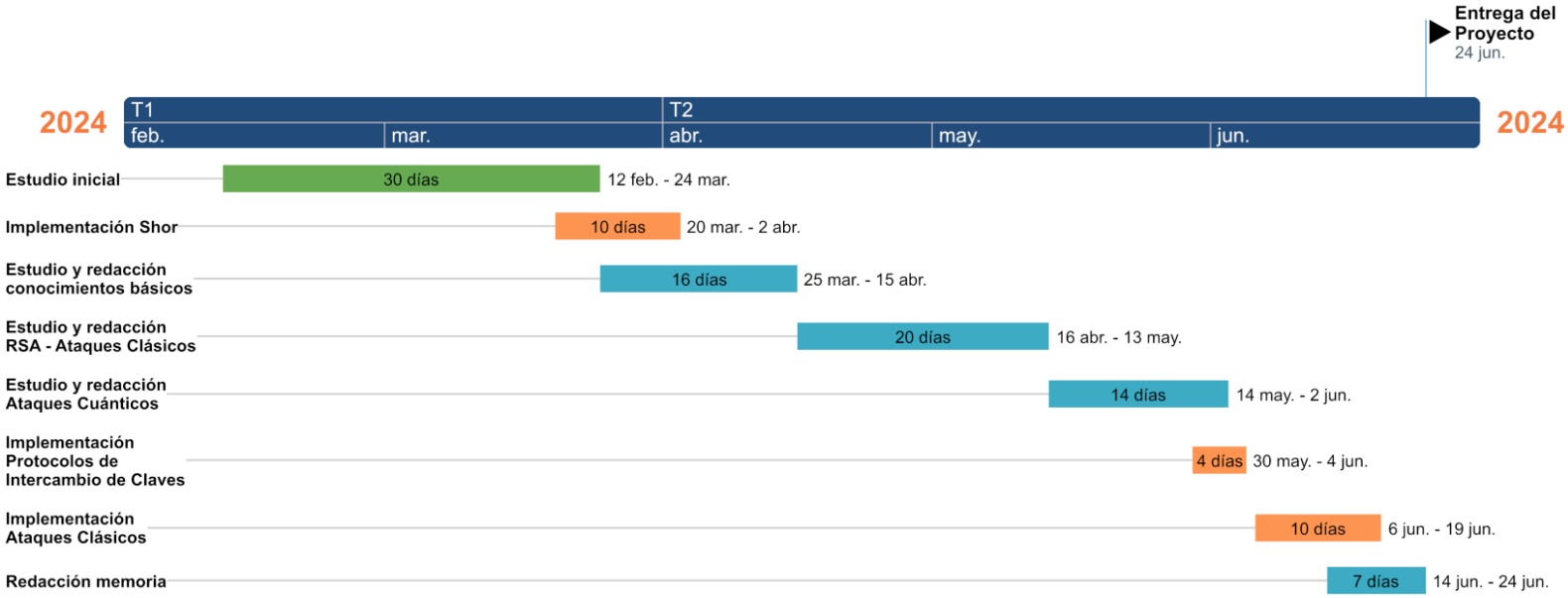

Figura 1.1: Planificación del trabajo.

1.4. Estructura

Este Trabajo Fin de Grado se estructura en cinco bloques en los que se organizan los diez capítulos que conforman esta memoria.

- **Bloque I. Introducción**. Contiene un único capítulo homónimo, donde se contextualiza el trabajo realizado, marca los objetivos a cumplir, propone una planificación inicial y presenta la estructura de la memoria.
- **Bloque II. Conocimientos básicos**. Explica las bases de las dos áreas del conocimiento principales de este trabajo. Se compone de dos capítulos:
  - *Introducción a la criptografía*: en él, definimos los principales conceptos en criptografía, como los diferentes tipos de cifrados y de ataques existentes.
  - *Introducción a la computación cuántica*: explica las bases sobre las que se sustenta la computación cuántica. Define conceptos importantes como qubit, puerta y circuito cuántico; y explica las leyes que determinan su funcionamiento.
- **Bloque III. Marco teórico**. Cubre el estudio relacionado con los objetivos concretos de este trabajo. Se compone de cuatro capítulos:
  - *Funcionamiento de RSA*: define las diferentes partes que componen RSA y realiza un criptoanálisis básico del mismo. Además, se cubren detalles teóricos de su implementación, como la generación de números primos.
  - *Ataques al RSA en computación clásica*: expone una serie de ataques que vulneran la seguridad de RSA. La mayoría de los algoritmos se enfocan en la factorización.
  - *Ataques cuánticos al RSA*: presenta el algoritmo de Shor en detalle y desarrolla una propuesta para su implementación.
  - *Protocolos de intercambio de claves por canales cuánticos*: en él, se plantean dichos protocolos y se estudia su funcionamiento.
- **Bloque IV. Implementación y experimentación**. Memoria de las implementaciones realizadas y de los resultados obtenidos a través de ellas.
  - *Implementación de RSA y ataques clásicos*: detalla el diseño y la programación de un programa que implementa el criptosistema RSA y algunos ataques sobre el mismo.

- *Implementación de algoritmos cuánticos*: desarrolla la memoria de la implementación práctica de algoritmos cuánticos y analiza los resultados de las mismas tras ser simuladas o ejecutadas en ordenadores cuánticos reales.

- **Bloque V. Conclusiones**. De un solo capítulo, expone un breve resumen de la ejecución del trabajo y de las posibles vías futuras que se pueden seguir a partir del mismo.

Parte II

Conocimientos básicos

Capítulo 2

Introducción a la criptografía

La criptografía, desde el punto de vista de la ingeniería, es la ciencia de cifrar y proteger mensajes mediante un algoritmo que, haciendo uso de diferentes herramientas matemáticas, lo modifica y garantiza su confidencialidad. De esta manera, a pesar de que el mensaje pueda ser interceptado en su envío por terceros, la criptografía se encarga de que únicamente los destinatarios autorizados puedan leerlo.

Por otro lado, también tenemos el *criptoanálisis*, que se encarga de la tarea contraria: trata de romper la confidencialidad de dichos mensajes. Ambas ciencias, criptografía y criptoanálisis, son complementarias: una vez se crea un algoritmo criptográfico, este debe ser examinado desde el punto de vista del criptoanálisis para poner a prueba su seguridad. Ambas ciencias conforman la llamada *criptología*.

Los mecanismos usados para proteger los mensajes en criptografía se denominan *criptosistemas*. Un criptosistema es descrito por los siguientes elementos:

1. M, que se define como el conjunto de los mensajes que puede cifrar.
2. C, el conjunto de todos los mensajes cifrados que son generados por el criptosistema, denominados *criptogramas*.
3. El conjunto de posibles claves de cifrado K_e y descifrado K_d que se pueden utilizar.
4. Un algoritmo de cifrado E, que dado un mensaje $m \in M$ y una clave de cifrado k_e, genera un criptograma $c \in C$. Puede no ser determinista, y que dado un mismo mensaje genere diferentes criptogramas.
5. Un algoritmo de descifrado D, que dado un criptograma $c \in C$ y la clave de descifrado k_d compañera a la clave de cifrado k_e usada para obtener el criptograma, obtiene el mensaje original m.

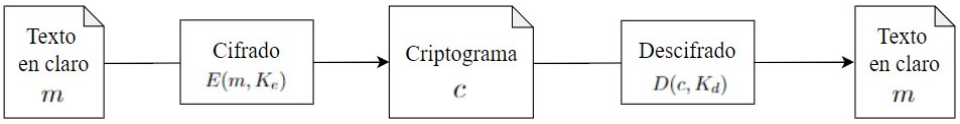

Figura 2.1: Esquema genérico de cifrado y descifrado.

Cabe destacar que no es lo mismo cifrar que codificar. Mientras que codificar un mismo mensaje devuelve siempre la misma secuencia y no pretende ocultar información, el cifrado genera diferentes criptogramas en función de los parámetros que especifiquemos al algoritmo de cifra y oculta el mensaje para que no lo puedan interpretar entidades que no conozcan dichos parámetros.

En 1883, el matemático A. Kerckhoffs definió unos principios que un criptosistema debe cumplir para ser seguro [Ker83]. A pesar de la antigüedad de los postulados, muchos de ellos siguen siendo válidos. Una adaptación de ellos a la criptografía moderna (esto es, a la introducción de los ordenadores en el proceso) son los siguientes principios:

1. El criptosistema debe ser en la práctica computacionalmente seguro, en caso de que no lo sea matemáticamente. Es decir, si el criptosistema presenta vulnerabilidades, explotarlas debe tener un coste computacional inviable.

2. El criptosistema no puede ser secreto: su seguridad no puede depender de que los criptoanalistas no puedan acceder a los algoritmos de cifrado y descifrado. De esta forma, la seguridad del mismo recae únicamente en la clave empleada.

3. Debe ser posible almacenar de forma segura las claves a emplear, así como comunicarlas con facilidad.

4. Los algoritmos de cifrado y descifrado deben ser sencillos y rápidos.

5. Los criptogramas deben ser fácilmente transmisibles.

Actualmente, los sistemas de cifra se clasifican en dos grandes grupos. En primer lugar, tenemos los cifrados simétricos o de clave secreta, los cuales emplean la misma clave para cifrar y descifrar ($k_e = k_d$). De esta manera, tanto emisor como receptor del criptosistema deben acordar una clave secreta a utilizar de antemano, lo cual puede ser poco conveniente.

En contraposición, tenemos los cifrados asimétrica o de clave pública, donde cada clave de cifrado k_e tiene su inversa k_d que recupera el mensaje original. Así, cada usuario puede publicar k_e, que en este caso llamaremos

clave pública, y mantener en secreto k_d, que será la *clave privada*. Cuando se quiere mandar un mensaje, el emisor emplea la clave pública del receptor para cifrar el mensaje, y el receptor lo descifra con su clave privada. Este enfoque resuelve el problema del intercambio de claves que veíamos en la cifra simétrica, pues los usuarios no necesitan acordar una clave secreta entre ellos.

2.1. Criptografía de clave secreta

Un criptosistema de clave secreta es una familia de pares de funciones $E_k : M \to C, D_k : C \to M$ para cada clave k perteneciente al espacio de claves K, tales que:

$$D_k(E_k(m)) = m.$$

Para enviar un mensaje secreto a través de este tipo de criptosistemas, el emisor y el receptor deben acordar usar una clave k con antelación. Entonces, el emisor cifra su mensaje $m \in M$ con la función de cifra E_k, y envía el criptograma c al receptor por el canal que utilicen:

$$E_k(m) = c \in C$$

Una vez el receptor recibe el criptograma c, lo descifra con la función D_k, consiguiendo el mensaje original:

$$D_k(c) = D_k(E_k(m)) = m$$

Si bien estos criptosistemas son muy eficientes y las claves que emplean son relativamente pequeñas, plantea ciertos problemas.

En primer lugar, los usuarios necesitan seleccionar una clave común, de forma que deben reunirse presencialmente para tal fin o transmitir la clave por un medio inseguro, lo cual puede comprometer la seguridad de la clave.

Además, precisa que cada par de usuarios que quieran comunicarse entre sí tengan una clave exclusiva, lo cual da lugar a un problema de almacenamiento de claves si el número de usuarios es muy alto.

Por último, este tipo de criptosistemas carecen de firma digital, por lo que no garantizan la autenticidad del mensaje. Al tener ambos usuarios la misma clave y compartir el algoritmo de cifra E_k, dado un criptograma $c = E_k(m)$ no podemos garantizar cuál de los usuarios ha escrito el mensaje [DHM05].

2.1.1. Cifrados en flujo

Los cifrados en flujo son algoritmos de cifrado simétrico que generan a partir de una clave k una secuencia de bits pseudoaleatorios, llamados *secuencia cifrante* (s_i). Así, generamos dicha cadena de bits y los sumamos modulo 2 a los bits que codifican el mensaje m, generando el criptograma c. Para descifrar c, el receptor procede de la misma manera: genera la secuencia de bits y la suma modulo dos con los bits del criptograma, obteniendo el mensaje original, pues:

$$c \oplus s_i = m \oplus s_i \oplus s_i = m \oplus 0 = m$$

Si el emisor y receptor poseen una secuencia cifrante secreta completamente aleatoria y emplean dicha secuencia una única vez para cifrar sus mensajes, la seguridad del mensaje es perfecta: ningún espía con potencia de cálculo infinita podrá saber nunca el mensaje original a partir del criptograma. Esto es lo que se conoce como cifrado con *libreta de un solo uso*. Este método, sin embargo, no resulta práctico, pues necesita almacenar claves del mismo tamaño que el mensaje, que solo se pueden usar una vez y que además deben acordar de antemano los interlocutores, lo cual requiere que ambos se citen presencialmente para compartir el secreto periódicamente.

En el capítulo 7, veremos una cura para estos males de la mano de la computación cuántica. Gracias a las propiedades de esta, se han diseñado protocolos que nos permiten compartir secuencias aleatorias de bits y asegurar que estas solo son conocidas por emisor y receptor. En consecuencia, esto posibilita obtener libretas de un solo uso de forma remota, haciendo factible y muy seguro el envío de información a través de este método.

Actualmente, los algoritmos de generación de secuencias cifrantes suelen implementarse haciendo uso de registros de desplazamiento realimentados linealmente (LFSR). Estos, a partir de un estado inicial del circuito, generan secuencias de bits pseudoaleatorias de gran tamaño. Algunos ejemplos de cifrados de flujo son A5/1 [EK03] y RC4 [Gar18].

2.1.2. Cifrados en bloque

Los cifrados de bloque cifran mensajes dividiéndolos en grupos de símbolos del mismo tamaño (bloques) y aplicando sobre cada uno de ellos un mismo algoritmo de cifra.

Estos algoritmos se componen de diferentes sistemas de cifrado elementales: sustituciones, transposiciones, transformaciones, etc. Con una combinación adecuada de estos, se busca la difusión y confusión del bloque original para generar un criptograma seguro. La difusión de un mensaje se trata de

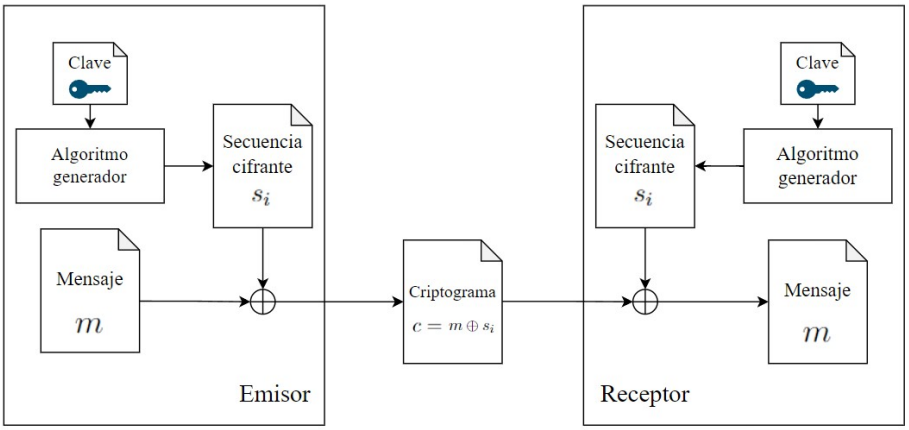

Figura 2.2: Cifrado en flujo.

redistribuir los caracteres originales del texto, empleando herramientas como la permutación de caracteres. La confusión, por otro lado, consiste en la modificación de los caracteres por otros distintos. Una forma sencilla de conseguir confusión en un texto es el cifrado César, que cambia cada letra del texto por aquella que esté cierto número de posiciones por delante en el alfabeto.

La forma en que se extiende el algoritmo de cifra E_K de cifrar bloques a cifrar mensajes de longitud arbitraria se denomina *modo de operación*. El más simple de ellos es el llamado *Electronic CodeBook* (ECB), en el que se divide el texto en claro en bloques y se cifra de forma independiente. Otro de ellos es el Cipher-Block Chaining (CBC). En él, se selecciona una segunda clave de forma aleatoria y se coloca como primer bloque del criptograma. Los siguientes bloques del criptograma se obtienen aplicando el algoritmo de cifra sobre la suma módulo 2 del bloque del texto en claro i y el bloque del criptograma i hasta cifrar todos los bloques de m. En las figuras 2.3 y 2.4 se ilustra el funcionamiento de estos dos modos de operación. Otros modos de operación conocidos son el Cipher FeedBack (CFB) o el Output FeedBack (OFB).

Además, aunque siempre se usa el mismo algoritmo para cifrar, es posible que no empleemos para cada bloque la misma clave k a la hora de aplicarlo. Si siempre empleamos la misma clave para cifrar, tenemos un cifrado *en cascada*, mientras que si usamos diferentes claves para cada bloque (obtenidas de una clave de mayor tamaño, a partir de un algoritmo de selección ordenada) tenemos un cifrado *en producto*.

Ejemplos famosos de algoritmos de cifrado en bloque son el *DES*, que fue estándar hasta su ruptura en 1998 [EFF98], o el *AES Rijndael* [DR98], sucesor del anterior, que es uno de los más usados actualmente.

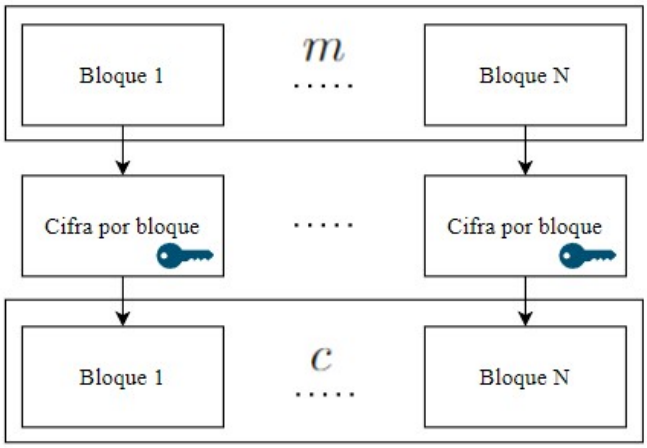

Figura 2.3: Cifrado en bloque en modo ECB.

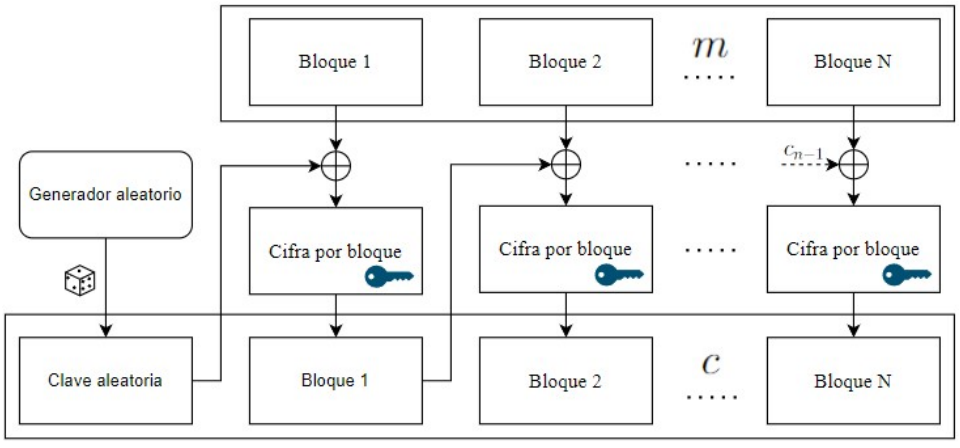

Figura 2.4: Cifrado en bloque en modo CBC.

2.2. Criptografía de clave pública

Al surgir las telecomunicaciones e internet en los años 70, se necesitaba un sistema para intercambio de claves seguro que no requiriera reunir a las partes interesadas. La criptografía de clave pública surge en este contexto para resolver las carencias de los criptosistemas de clave secreta.

2.2.1. Protocolo de Intercambio de Diffie-Hellman

En 1976, se publicó el *Protocolo de Intercambio de Diffie-Hellman* [DH76]. Este algoritmo permite generar información secreta mediante un canal inseguro. El protocolo consta de los siguientes pasos:

1. Los dos usuarios que usan el protocolo, que llamaremos *Alice* (A) y *Bob* (B), eligen públicamente un grupo[1] cíclico[2] finito G de orden[3] n, así como un generador α de dicho grupo. Usualmente, se usa $G = \mathbb{Z}_p$ con p primo y $|\mathbb{Z}_p| = p - 1$.
2. A genera un número aleatorio a y calcula $x = \alpha^a$ en G. Una vez calculado, envía x a B.
3. B genera otro número aleatorio b y calcula $y = \alpha^b$ en G. Luego, envía y a A.
4. A, al recibir de B el valor de y, calcula $y^a = (\alpha^b)^a$ en G.
5. Igualmente, cuando B recibe de A el valor de x, calcula $x^b = (\alpha^a)^b$ en G.

Una vez realizados estos pasos, A y B comparten como clave secreta:

$$(\alpha^a)^b = y^a = (\alpha^b)^a = \alpha^{ab}$$

La clave del algoritmo es que un tercero que observa la comunicación entre A y B solo conoce G, n, α, α^a y α^b para calcular α^{ab}. No obstante, este algoritmo no permite a ninguno de los actores controlar cuál será el valor de la clave generada al ejecutar el protocolo. Actualmente, no existe ningún algoritmo en tiempo polinómico que, a partir de dichos datos, sea capaz de resolver este problema. Este es el llamado *Problema de Diffie-Hellman* (PDH).

[1] Un grupo $(G, *)$ es un conjunto de elementos donde se puede realizar una operación $*$ interna (siempre da como resultado un elemento $a \in G$), asociativa, con elemento neutro e inversa para todo elemento del grupo.

[2] Un grupo es cíclico si existe un elemento $\alpha \in G$ tal que todo elemento de G se puede escribir como potencia de G. A dicho elemento α se le conoce como *generador*.

[3] El orden de un grupo es el número de elementos que tiene.

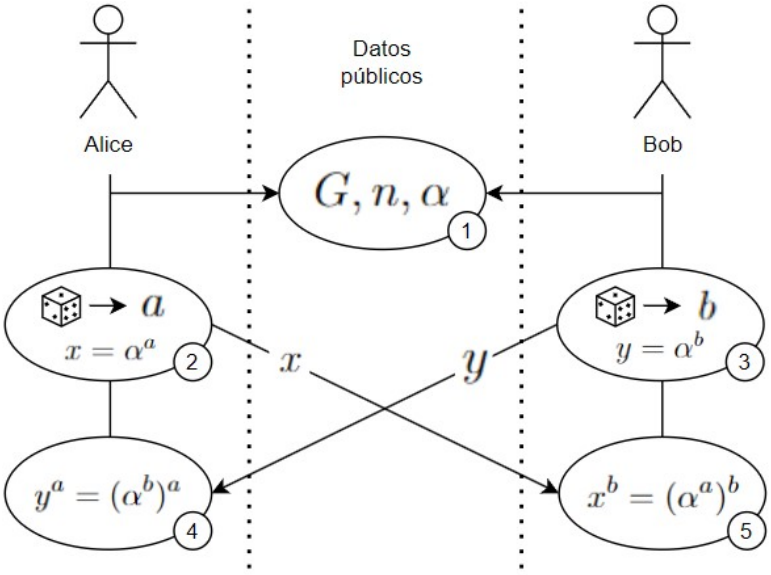

Figura 2.5: Protocolo de Intercambio de Diffie-Hellman

El PDH es un problema muy similar al *Problema del logaritmo discreto* (PLD), que plantea encontrar a dados G, n, α y α^a. Si se conociese un algoritmo eficiente para resolver el PLD, el PDH también estaría resuelto, pues se podría calcular a usando dicho algoritmo y luego calcular $(\alpha^b)^a$.

2.2.2. Protocolos de cifrado de clave pública

Poco después de la publicación del intercambio de clave de Diffie-Hellman, comenzaron a surgir los primeros criptosistemas de clave pública.

Un criptosistema de clave pública se define como un par de familias de funciones de cifrado E_e y descifrado D_d con claves de cifrado e y descifrado d, tales que:

$$\forall m \in M, D_d(E_e(m)) = D_d(c) = m$$

De esta manera, cada usuario tiene dos claves: una pública e y una privada d. Si algún emisor quiere mandar un mensaje a un usuario, este lo cifrará empleando el algoritmo con la clave pública del receptor:

$$c = E_e(m)$$

Una vez el receptor recibe el criptograma c, lo descifra aplicando sobre este el algoritmo con su clave privada D_d, consiguiendo el mensaje original:

$$D_d(c) = m$$

La seguridad de estos criptosistemas radica en la dificultad de descifrar el criptograma c sin conocer la clave privada del destinatario y de calcular la clave privada a partir del conocimiento de la clave pública. Para diseñarlos, suelen usarse *funciones unidireccionales* (OWF[4]): funciones que son rápidas de calcular para cualquier entrada, pero cuya función inversa es computacionalmente demasiado costosa como para poder calcularse.

Otra opción son las funciones unidireccionales con trampilla (TOWF[5]), que son también fáciles de calcular y tienen una función inversa cuyo cálculo es intratable, con la diferencia de que, si se dispone de cierta información adicional (*trampilla*), es factible su cálculo.

No se ha demostrado matemáticamente la existencia de estos tipos de funciones, pero actualmente hay bastantes funciones aspirantes a pertenecer a dichos conjuntos.

Los criptosistemas de este tipo más conocidos son *RSA*[RSA78], el cual describiremos más adelante, *ElGamal*[Elg85] y *DSA* [SN98].

Aunque este tipo de criptosistemas presentan varias ventajas, tienen el inconveniente de ser más lentos que los de clave secreta. Además, son susceptibles a ataques de tipo *Man-in-the-middle*: cuando el emisor consulta la clave pública del destinatario, un tercero puede interceptar la petición y mandar entonces su clave pública. Así, el emisor pasará el mensaje a dicho tercero y el tercero podrá descifrarlo con su clave privada, y después pasarlo al destinatario real. Es por esto que los criptosistemas de clave pública se apoyan en "Terceras Partes de Confianza" (TTPs[6]) y en el uso de certificados digitales.

2.3. Seguridad y ataque de sistemas criptográficos

La seguridad de un criptosistema consiste en la dificultad que tienen entidades no autorizadas de vulnerar la confidencialidad, integridad, autenticidad o disponibilidad de los mensajes que se envían a través de este.

Los ataques criptográficos son cualquier intento de comprometer la seguridad de un sistema criptográfico. Estos ataques pueden dirigirse contra diversos aspectos de los sistemas criptográficos, como algoritmos de cifrado, protocolos de comunicación segura, entre otros.

Según la actitud del atacante, podemos clasificarlos en dos grupos: ataques pasivos y ataques activos. Los ataques pasivos son aquellos en los que el atacante solamente observa los datos enviados de un usuario a otro, sin

[4] *One-Way Functions*, en inglés
[5] *Trapdoor One-Way Functions*, en inglés.
[6] Del inglés, *Trusted Third Party*.

interceptarlos ni modificarlos. Este tipo de ataques atenta solo contra la confidencialidad de los mensajes. Entre ellos, según la información y herramientas de las que disponga el criptoanalista, podemos distinguir:

1. *Ataque de fuerza bruta*: el criptoanalista prueba todas las posibles claves y observa si alguna de ellas le devuelve un mensaje claro. Si el espacio de claves es demasiado pequeño, este ataque trivial lo romperá.

2. *Ataque al texto cifrado*: en él, el adversario trata de descifrar un criptograma c sin ningún conocimiento adicional. Si un criptosistema es susceptible a este tipo de ataque, se considera completamente inseguro. Si no, consideraremos que el criptosistema es *unidireccional con respecto a ataques pasivos*.

3. *Ataque al texto claro conocido*: en este caso, el criptoanalista dispone de algunos pares de mensajes con sus correspondientes criptogramas.

4. *Ataque al texto claro elegido*: aquí, el atacante puede obtener el texto cifrado correspondiente a un texto en claro de su elección. A partir de esto, el criptoanalista puede buscar cómo descifrar los criptogramas o averiguar la clave. Si un criptosistema es resistente a ellos, se denomina *unidireccional con respecto a ataques de texto en claro escogido*.

5. *Ataque al texto cifrado elegido*: el adversario puede obtener el texto claro correspondiente a un texto cifrado de su elección, con la excepción de que no puede elegir el criptograma objetivo que pretende descifrar. Si un criptosistema es resistente a ellos, se denomina *unidireccional con respecto a ataques de criptograma escogido*.

6. *Ataque adaptable al texto claro/cifrado elegido*: equivalente a los dos anteriores, pero el atacante puede obtener todos los textos cifrados o claros, respectivamente, que quiera de su elección (excepto, de nuevo, usar el descifrado para descifrar el criptograma objetivo). Si un criptosistema es resistente a ellos, se denomina *polinomialmente seguro contra ataques de texto en claro/criptograma escogido*.

Por otro lado, en los ataques activos el adversario altera los mensajes entre los usuarios, poniendo en riesgo la confidencialidad, integridad y autenticidad de la información. Los principales son los siguientes:

1. *Man-in-the-middle*: el adversario se coloca entre la comunicación de los usuarios, pudiendo interceptar los mensajes de uno a otro y sustituirlos por sus propios mensajes.

2. *Ataques de canal lateral*: el criptoanalista obtiene información útil a partir de medidas físicas de ejecución del sistema, como el consumo de tiempo o energía del mismo.

Capítulo 3

Introducción a la computación cuántica

La computación cuántica nace a principios de los años 80 de la mano de los físicos Paul Benioff [Ben80] y Richard Feynman [Fey82], quienes propusieron aprovechar las propiedades del mundo subatómico para la realización de cálculos en computadores. Desde entonces, muchos investigadores han estudiado en profundidad este campo y han hecho descubrimientos relevantes sobre cómo esta tecnología puede resolver problemas que con la computación clásica resultaban inabordables. Para adentrarnos en este fascinante campo, seguiremos como referencias principales el reconocido libro [NC10] y [Won22].

3.1. Qubit

El qubit[1] es la unidad mínima de información que maneja un ordenador cuántico. Mientras que un bit clásico solamente puede estar en dos estados (0 y 1), el estado de un qubit se rige por el *principio de superposición cuántica*: puede adoptar el valor 0, 1, o una "mezcla" de ambos.

Matemáticamente, un qubit $|\psi\rangle$[2] suele representarse como una combinación lineal de los estados correspondientes a los estados clásicos 0 y 1 en computación cuántica, que son $|0\rangle$ y $|1\rangle$:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = \begin{pmatrix}\alpha\\ \beta\end{pmatrix} \tag{3.1}$$

donde $\alpha, \beta \in \mathbb{C}$ y $|\alpha|^2 + |\beta|^2 = 1$.

[1] Apócope de Quantum Bit.

[2] La notación $|x\rangle$ (leído "ket x") es la llamada *Notación de Dirac*, normalmente usada para representar estados en mecánica cuántica.

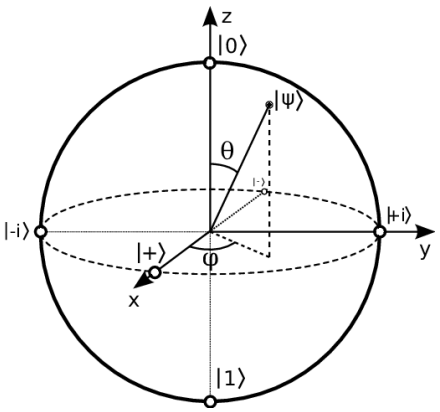

Figura 3.1: Esfera de Bloch y la ubicación de los estados $|0\rangle$, $|1\rangle$, $|+\rangle$, $|-\rangle$, $|+i\rangle$ y $|-i\rangle$ [PA22].

Por ello, otra representación típica de un qubit es un vector unitario en un espacio bidimensional, donde $|0\rangle$ y $|1\rangle$ forman una base ortonormal.

Una peculiaridad de la computación cuántica es que, cuando medimos un qubit, podemos obtener el qubit $|0\rangle$ con probabilidad $|\alpha|^2$ o el qubit $|1\rangle$ con probabilidad $|\beta|^2$. Una vez medido el qubit, su estado queda definido y se pierde la superposición de estados en la que se encontraba anteriormente, por lo que si volvemos a medir el qubit obtendremos de nuevo el mismo resultado.

Una representación más visual de un qubit es la *Esfera de Bloch* (Figura 3.1), cuyos puntos componen todos los estados posibles de un qubit. La posición de un qubit en la esfera se define mediante dos parámetros θ y ϕ, que se obtienen a partir de la representación polar[3] de los coeficientes α, β:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle = r_0 e^{i\phi_0}\,|0\rangle + r_1 e^{i\phi_1}\,|1\rangle = e^{i\phi_0}(r_0\,|0\rangle + r_1 e^{i(\phi_1-\phi_0)}\,|1\rangle)$$

donde $r_0, r_1 \in \mathbb{R}$ y $\phi_0, \phi_1 \in [0, 2\pi]$. El coeficiente $e^{i\phi_0}$ se conoce como fase global, y resulta irrelevante a la hora de aplicar las mediciones sobre el qubit. Tomando $\phi = \phi_1 - \phi_0$, nos queda:

$$|\psi\rangle = r_0\,|0\rangle + r_1 e^{i\phi}\,|1\rangle$$

Para obtener el parámetro $\theta \in [0, \pi]$, aplicamos la sustitución $r_0 = cos\frac{\theta}{2}$ y $r_1 = sen\frac{\theta}{2}$, la cual es posible hacer gracias a que $|r_0|^2 + |r_1|^2 = 1$. Por tanto, llegamos a la expresión:

$$|\psi\rangle = cos\frac{\theta}{2}\,|0\rangle + sen\frac{\theta}{2}e^{i\phi}\,|1\rangle$$

[3]Un número complejo puede representarse como $re^{\alpha i}$, donde α es el ángulo que forma el eje X con la línea que une el punto en el plano complejo con el origen de coordenadas, y r es la distancia de dicho punto al origen. Esta es la llamada *forma polar* del mismo.

De esta forma, podemos representar cualquier estado de un qubit en la esfera. El parámetro θ nos indicará si el qubit tiene una componente α o β más fuerte, mientras que ϕ señalará la fase compleja que existe entre ambas.

En el ecuador de la esfera, tenemos todos estados de los qubits que, al medirse, colapsarán al estado $|0\rangle$ o $|1\rangle$ con igual probabilidad. De entre estos qubits, podemos destacar aquellos que coinciden con los ejes x e y en la esfera de Bloch:

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \qquad |-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \qquad |i\rangle = \frac{|0\rangle + i\,|1\rangle}{\sqrt{2}} \qquad |-i\rangle = \frac{|0\rangle - i\,|1\rangle}{\sqrt{2}}$$

A pesar de que un qubit pueda presentarse en infinitos estados, siempre nos encontraremos con el inconveniente de que perderemos su superposición de estados al medirlo. Por tanto, es preciso preparar infinitos qubits de forma idéntica para poder obtener α y β. Sin embargo, mientras no midamos los qubits, el hecho de poder trabajar con superposiciones de estados hace que podamos lidiar con una gran cantidad de "información oculta", la cual aumenta exponencialmente en función del número de qubits en el sistema. Es en este hecho donde reside el potencial de la computación cuántica frente a la clásica.

3.2. Sistemas de más de un qubit

Tal y como en un computador clásico el número posible de estados es $2^{N_{bits}}$, en un sistema cuántico de N qubits tendremos 2^N estados básicos. Matemáticamente hablando, el estado básico de N qubits viene dado por el producto tensorial de N qubits, representados como $|\psi\rangle = \begin{pmatrix}\alpha\\ \beta\end{pmatrix}$.

De esta manera, podemos representar el estado de un sistema de varios qubits como la superposición de los diferentes estados en los que se pueden encontrar, cada uno multiplicado por un coeficiente complejo α_s que define la probabilidad de que al medir el sistema los qubits converjan a dicho estado (que sería, concretamente, $|\alpha_s|^2$). A dichos coeficientes se les conoce como *amplitudes*.

Por ejemplo, dado un sistema de dos qubits, su estado vendrá dado por:

$$|\psi_0\rangle\,|\psi_1\rangle = \begin{pmatrix}\alpha_0\\ \beta_0\end{pmatrix} \otimes \begin{pmatrix}\alpha_1\\ \beta_1\end{pmatrix} = \begin{pmatrix}\alpha_0 \begin{pmatrix}\alpha_1\\ \beta_1\end{pmatrix}\\ \beta_0 \begin{pmatrix}\alpha_1\\ \beta_1\end{pmatrix}\end{pmatrix} = \begin{pmatrix}\alpha_0\alpha_1\\ \alpha_0\beta_1\\ \beta_0\alpha 1\\ \beta_0\alpha_1\end{pmatrix} = \begin{pmatrix}\alpha_{00}\\ \alpha_{01}\\ \alpha_{10}\\ \alpha_{11}\end{pmatrix} \tag{3.2}$$

cumpliendo siempre que $\sum_{i\in\{0,1\}^2} |\alpha_i|^2 = 1$.

Aquí entra en juego otra de las características que diferencia la computación cuántica de la clásica: el **entrelazamiento cuántico**. Este consiste en que si se miden parte de los qubits de un sistema, las probabilidades en las medidas posteriores del resto de qubits se ven alteradas.

Supongamos que, en el sistema de dos qubits $|\psi\rangle$ anterior, medimos el primero de los qubits y obtenemos $|0\rangle$. Entonces, el sistema se ve alterado y estaríamos en el siguiente estado:

$$|\psi\rangle = \frac{\alpha_{00}\,|00\rangle + \alpha_{01}\,|01\rangle}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}} = |0\rangle\,\frac{\alpha_{00}\,|0\rangle + \alpha_{01}\,|1\rangle}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}} \tag{3.3}$$

Dado que las probabilidades deben sumar 1, normalizamos por la probabilidad de haber obtenido un 0 en la primera medición, que es $\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}$. Así, vemos que la probabilidad de medir $|0\rangle$ en el segundo qubit se ha visto modificada (antes era $|\alpha_{00}|^2 + |\alpha_{10}|^2$, mientras que ahora es $\frac{|\alpha_{00}|^2}{\sqrt{|\alpha_{00}|^2+|\alpha_{01}|^2}}$).

Esto es, las mediciones de los qubits están *correladas*: el resultado de una influye en los potenciales resultados de las siguientes.

El máximo exponente de este fenómeno es el *estado de Bell* o *par EPR*, que se describe como sigue:

$$|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} \tag{3.4}$$

En este estado, tenemos que el resultado de una medición en un qubit definirá el resultado de la medición del otro qubit, de forma que el resultado en la segunda medición siempre será igual que el de la primera. Este estado es protagonista en muchos circuitos cuánticos conocidos, como en la teleportación cuántica [Ben+93] y el *superdense coding* [BW92].

3.3. Circuitos cuánticos

De forma paralela a la computación clásica, la forma en que se modifican los estados cuánticos de un ordenador cuántico se representa mediante circuitos y puertas cuánticas.

Un circuito cuántico (figura 3.2) se compone de dos tipos de elementos:

- **Cables**, donde cada cable representa un qubit a lo largo de la ejecución del circuito.
- **Puertas**, las cuales modifican el estado de uno o varios qubits. Estas se colocan sobre los cables que representan a los qubits sobre los que

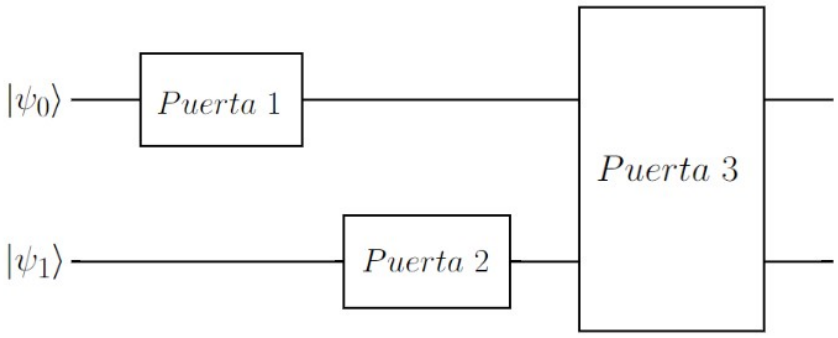

Figura 3.2: Esquema de ejemplo de circuito cuántico.

actúan, dibujando a la izquierda aquellas puertas que actúan en primer lugar.

Este tipo de circuitos presenta una serie de diferencias con los clásicos. En primer lugar, vemos que representan el orden de aplicación de las puertas sobre los qubits dibujándolas de izquierda a derecha. Debido a ello, no podemos formar ciclos en un circuito cuántico, pues ello representaría que estamos empleando una entrada actual para modificar el valor que tuvo en el pasado otro qubit, lo cual no tiene sentido. Además, los circuitos clásicos permiten juntar cables, realizándose una operación 'or'. Esto tampoco se permite en los circuitos cuánticos, pues supondría juntar dos qubits en uno solo, lo cual no es una operación válida. Tercero, tampoco podemos dividir cables, pues clonar qubits sin saber a priori en qué estado se encuentran es una operación imposible en computación cuántica, según estipula el *Teorema de no clonación.*

3.3.1. Puertas cuánticas

A diferencia de las puertas lógicas clásicas, que podemos definir mediante una tabla de verdad para cada una de las posibles entradas, las puertas cuánticas se definen como una función cuyo dominio abarca los infinitos posibles qubits de entrada.

Además, debido al funcionamiento de la mecánica cuántica, las puertas efectuarán cambios lineales en las entradas. Las puertas cuánticas de un qubit suelen representarse, consecuentemente, como matrices de tamaño 2×2, siendo N el número de qubits de entrada. Otra representación típica es especificar la acción que tiene la puerta sobre los estados elementales $|0\rangle$ y $|1\rangle$ y aplicar linealidad: $f(\alpha\,|0\rangle + \beta\,|1\rangle) = \alpha f(|0\rangle) + \beta f(|1\rangle)$.

La única condición para que una matriz sea una puerta cuántica válida es que, al multiplicarse por N qubits cualesquiera, el resultado presente N

qubits válidos. Visto de otra forma, las puertas lógicas siempre deben devolver un estado normalizado, tal que la suma de probabilidades de obtener cada estado básico en el mismo sea 1. Esta propiedad es la que define las *matrices unitarias*, caracterizadas por cumplir que $A^\dagger A = I$, siendo $A^\dagger$ la matriz traspuesta conjugada de A.

No obstante, también existe otra propiedad muy deseable para las puertas cuánticas: la *reversibilidad*. Una puerta es reversible si a partir de cualquiera de sus salidas se puede reconstruir la entrada. Por ejemplo, la puerta clásica AND no es reversible, pues si la salida es 0 existen tres posibles estados de entrada: 00, 01 y 10, y nunca sabremos cual de ellos fue el usado.

El interés de esta propiedad viene dado por la energía que usan los circuitos. Esta conexión viene reflejada en el principio de Landauer [Lan61]:

Principio de Landauer. Supongamos que un ordenador borra un bit de información. La cantidad de energía disipada en el entorno es, al menos, $k_B T ln\ 2$, siendo k_B la constante de Boltzmann y T la temperatura en el entorno.

Entonces, si empleamos puertas irreversibles, sabemos que parte de la información del sistema se pierde, y esto ocasiona el consumo de energía. Por contra, si somos capaces de emplear únicamente puertas cuánticas reversibles, nunca se borra información y el costo energético del circuito cuántico se reduce a cero. En consecuencia, podemos ver el circuito cuántico como un sistema cerrado que una vez inicializado no requiere que le suministremos energía externa. No obstante, aunque esto ayuda, cabe destacar que el ordenador cuántico aún tendrá un elevado consumo energético por el resto de sus elementos, como la refrigeración, electrónica, etc.

Puertas cuánticas de un qubit

Existen infinitas matrices unitarias que se corresponden con puertas cuánticas válidas de un qubit. A continuación, veremos las más comunes:

- **Puerta Pauli-X**: podemos considerarlo como el equivalente a la puerta NOT en computación cuántica: intercambia los estados $|0\rangle$ y $|1\rangle$ entre sí. Debe su nombre a que equivale a una rotación de 180^{o} del qubit en el eje X en la esfera de Bloch.

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \qquad \begin{matrix} |0\rangle \longrightarrow |1\rangle \\ |1\rangle \longrightarrow |0\rangle \end{matrix} \tag{3.5}$$

- **Puerta Pauli-Z**: esta puerta cambia el signo de la componente β que acompaña a $|1\rangle$, lo cual hace que el qubit cambie de fase. Al aplicarla,

el qubit rota 180^{o} en el eje Z en la esfera de Bloch.

$$Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad \begin{array}{l} |0\rangle \longrightarrow |0\rangle \\ |1\rangle \longrightarrow -|1\rangle \end{array} \tag{3.6}$$

- **Puerta H (de *Hadamard*)**: esta puerta lleva el estado $|0\rangle$ a $|+\rangle$ y el estado $|1\rangle$ a $|-\rangle$. En este caso, su acción se puede interpretar como una rotación de 90^{o} alrededor del eje Y, seguida de una de 180^{o} en el eje X.

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad \begin{array}{l} |0\rangle \longrightarrow |+\rangle \\ |1\rangle \longrightarrow |-\rangle \end{array} \tag{3.7}$$

En general, toda puerta cuántica de un qubit puede descomponerse en tres rotaciones de la siguiente manera:

$$U = e^{i\alpha} \begin{bmatrix} e^{-i\beta/2} & 0 \\ 0 & e^{i\beta/2} \end{bmatrix} \begin{bmatrix} cos\frac{\gamma}{2} & -sen\frac{\gamma}{2} \\ sen\frac{\gamma}{2} & cos\frac{\gamma}{2} \end{bmatrix} \begin{bmatrix} e^{-i\delta/2} & 0 \\ 0 & e^{i\delta/2} \end{bmatrix} \tag{3.8}$$

donde α denota una fase global (la cual se suele ignorar, pues no afecta a los cálculos/mediciones), y β, γ, δ son los ángulos de giro del qubit respecto a los ejes Z, Y, y de nuevo Z, respectivamente.

3.3.2. Puertas cuánticas de varios qubit

Para poder realizar cálculos más allá de girar un qubit, como el entrelazamiento cuántico o las operaciones condicionales, necesitamos puertas que actúen sobre dos o más qubits.

La puerta más famosa que cumple esta función es la **puerta CNOT**. La acción de esta puerta equivale a aplicar una puerta X sobre el segundo qubit (que llamaremos *objetivo*) si el primer qubit (*de control*) vale $|1\rangle$, y no hacer nada si el primero tiene el valor $|0\rangle$.

Otra manera de ver su funcionamiento es comparándolo con la suma módulo 2, de forma que $\text{CNOT}(|xy\rangle) = |x\ x \oplus y\rangle$. Es por esto que en los circuitos cuánticos se suele representar con el símbolo "$\oplus$".

Otra puerta importante es el **CCNOT o puerta de Toffoli**. Esta puerta actúa sobre tres qubit (dos de control y uno objetivo), y su acción es aplicar una puerta X al qubit objetivo si y solo si los qubits de control valen $|11\rangle$.

3.3.3. Medición

En computación cuántica, la medición también se representa como una puerta en los circuitos cuánticos, pues esta supone una alteración del estado

del sistema. Esta puerta suele representarse con el icono de un medidor.

Una vez se mide un qubit (lo cual se suele realizar sobre el eje Z), su estado converge a $|0\rangle$ o $|1\rangle$, convirtiéndose en información clásica: un bit. Para denotar bits clásicos en un circuito cuántico, usaremos una doble línea.

Realmente, uno puede medir un qubit en cualquiera de los ejes de la esfera de Bloch, y no solamente en el eje Z dando como resultado $|0\rangle$ o $|1\rangle$. Al realizar una medida sobre un eje, el qubit convergerá a alguno de los dos polos de dicho eje. Cuanto más próximo esté el qubit de dicho polo al medirse, mayor será la probabilidad de que este colapse al mismo. Para calcular matemáticamente las probabilidades exactas de que un qubit converja a un estado u otro de una base, podemos cambiar la representación del qubit a medir a dicha base y calcular los coeficientes de cada estado básico de la nueva base. Entonces, nos bastará con tomar el cuadrado del módulo de los coeficientes para hallar la probabilidad de cada estado.

3.3.4. Ejemplo: Teleportación cuántica

La teleportación cuántica es un fenómeno en el cual se realiza una transferencia de información de un lugar a otro, sin que la información pase físicamente por el espacio entre ellos. Esto lo podemos conseguir con el siguiente circuito:

Así, vemos cómo un qubit $|\psi\rangle$, desconocido, es llevado del primer qubit

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

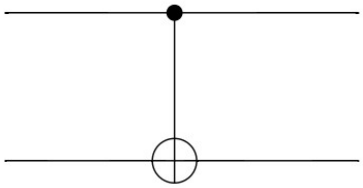

Figura 3.3: Puerta CNOT y representación en un circuito cuántico. El cable de arriba representa el qubit de control, y el de abajo el qubit objetivo.

$$CCNOT = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

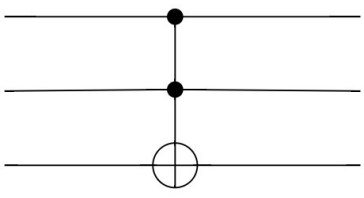

Figura 3.4: Puerta de Toffoli y representación en un circuito cuántico.

Figura 3.5: Medición en un circuito cuántico.

al tercero sin que dicho qubit actúe directamente sobre el estado del qubit destino.

Para entender por qué esto funciona, vamos a analizar este circuito paso a paso:

1. Inicialmente, comenzamos con el estado $|\psi 00\rangle$, siendo $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$, con α y β desconocidos. Así, podemos expresar nuestro estado inicial con la siguiente fórmula:

$$S_0 = \alpha\,|000\rangle + \beta\,|100\rangle$$

2. Luego, aplicamos una puerta H sobre el tercer qubit. Esto hará que dicho qubit pase del estado $|0\rangle$ al estado $|+\rangle = \frac{1}{\sqrt{2}}\,|0\rangle + |1\rangle$. Por tanto, el estado global en este punto es:

$$S_1 = \frac{\alpha\,|000\rangle + \alpha\,|001\rangle + \beta\,|100\rangle + \beta\,|101\rangle}{\sqrt{2}}$$

3. Lo siguiente que se hace es aplicar una CNOT con control en el qubit 3 y acción en el 2. Con esto, lo que conseguimos es que los qubits 2 y 3 tengan el mismo estado, conformando un *estado de Bell* entre ellos:

$$S_2 = \frac{\alpha\,|000\rangle + \alpha\,|011\rangle + \beta\,|100\rangle + \beta\,|111\rangle}{\sqrt{2}}$$

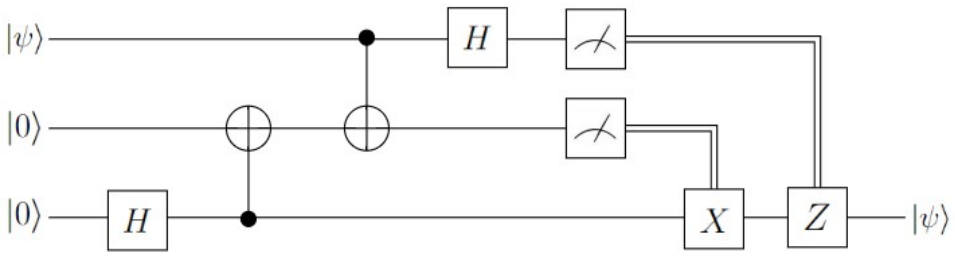

Figura 3.6: Teleportación cuántica.

4. Posteriormente, se aplica una puerta CNOT del qubit 1 al qubit 2, resultando en el siguiente estado:

$$S_3 = \frac{\alpha\,|000\rangle + \alpha\,|011\rangle + \beta\,|110\rangle + \beta\,|101\rangle}{\sqrt{2}}$$

5. Antes de medir, aplicamos una puerta H sobre el primer qubit, llevándonos al siguiente estado:

$$S_4 = \frac{\alpha\,|+00\rangle + \alpha\,|+11\rangle + \beta\,|-10\rangle + \beta\,|-01\rangle}{\sqrt{2}}$$

Pasando todo a los estados básicos $|0\rangle$ y $|1\rangle$, y posteriormente reagrupando términos en función del valor de los dos primeros qubits, tenemos:

$$S_4 = \frac{1}{\sqrt{2}}(\alpha\frac{|0\rangle + |1\rangle}{\sqrt{2}}\,|00\rangle + \alpha\frac{|0\rangle + |1\rangle}{\sqrt{2}}\,|11\rangle + \beta\frac{|0\rangle - |1\rangle}{\sqrt{2}}\,|10\rangle + \beta\frac{|0\rangle - |1\rangle}{\sqrt{2}}\,|01\rangle) =$$

$$= \frac{1}{\sqrt{2}}(\alpha\frac{|000\rangle + |100\rangle}{\sqrt{2}} + \alpha\frac{|011\rangle + |111\rangle}{\sqrt{2}} + \beta\frac{|010\rangle - |110\rangle}{\sqrt{2}} + \beta\frac{|001\rangle - |101\rangle}{\sqrt{2}}) =$$

$$= \frac{1}{2}(|00\rangle\,(\alpha\,|0\rangle + \beta\,|1\rangle) + |01\rangle\,(\beta\,|0\rangle + \alpha\,|1\rangle) + |10\rangle\,(\alpha\,|0\rangle - \beta\,|1\rangle) + |11\rangle\,(\beta\,|0\rangle - \alpha\,|1\rangle))$$

Si observamos esta expresión, podemos ver en qué estado quedaría el tercer qubit en función de los estados del primer y segundo qubit.

6. Una vez medimos los dos primeros qubits en el anterior estado, nos podemos encontrar en cuatro situaciones:

   - Si el resultado es 00, entonces el tercer qubit estará en el estado

$$\alpha\,|0\rangle + \beta\,|1\rangle = |\psi\rangle$$

   - Si el resultado es 01, entonces el tercer qubit estará en el estado

$$\beta\,|0\rangle + \alpha\,|1\rangle = X(|\psi\rangle)$$

   - Si el resultado es 10, entonces el tercer qubit estará en el estado

$$\alpha\,|0\rangle - \beta\,|1\rangle = Z(|\psi\rangle)$$

   - Si el resultado es 11, entonces el tercer qubit estará en el estado

$$\beta\,|0\rangle - \alpha\,|1\rangle = Z(X(|\psi\rangle))$$

Por tanto, para que en todos los casos tengamos en el tercer qubit el estado $|\psi\rangle$, aplicamos sobre el mismo una puerta X si el bit 2 tiene valor 1, y posteriormente una puerta Z si el bit 1 tiene valor 1.

La teleportación cuántica es solo uno de los extraños resultados que esta ciencia posee. Otros algoritmos básicos de interés que ilustran el potencial de la computación cuántica son el algoritmo de Bernstein-Varizani [BV97], donde se obtiene una clave a través de una única pregunta a un oráculo, cuya versión clásica necesitaría N preguntas; y el algoritmo de Deutsch-Jozsa [DJ92], capaz de distinguir entre funciones constantes o balanceadas (es decir, que su imagen consta de dos valores y ambos aparecen con la misma frecuencia) ejecutando una única vez la función.

Este tipo de aplicaciones suponen una gran mejora frente a los ordenadores convencionales en términos de eficiencia computacional. Sin embargo, encontrar algoritmos cuánticos que consigan superar en eficiencia a los algoritmos clásicos en tareas de aplicación real no es una tarea trivial, y miles de investigadores se dedican a tiempo completo en buscar dichos algoritmos.

En el Capítulo 6, veremos cómo podemos aprovechar los fenómenos cuánticos para construir circuitos que nos ayuden a vulnerar el criptosistema RSA.

Parte III

Marco teórico

Capítulo 4

Funcionamiento de RSA

4.1. Algoritmo RSA

El criptosistemas RSA consiste en un criptosistema de clave pública de cifrado de bloque. Este se divide en tres partes: una primera fase donde se generan las claves pública y privada del usuario, una segunda de cifrado del mensaje, y finalmente la fase de descifrado. A continuación, vamos a abordar cada una de estas fases en detalle, siguiendo principalmente [DHM05].

4.1.1. Generación de claves

Para generar las claves de un usuario U, aplicamos el siguiente algoritmo:

1. Elegimos dos primos, p y q, y calculamos su producto $n = pq$ y su función de Euler[1] $\phi(n) = (p-1)(q-1)$.[2].
2. Posteriormente, elegimos un entero positivo $e : 2 < e < \phi(n)$ tal que $mcd(e, \phi(n) = 1)$.
3. Luego, calculamos el inverso de e en el grupo multiplicativo $\mathbb{Z}^*_{\phi(n)}$, esto es, calculamos d tal que $e \cdot d \equiv 1 \ (mod\ \phi(n))$. Conociendo $\phi(n)$, d se puede conseguir con el Algoritmo Extendido de Euclides.
4. La clave pública del usuario es el par (n, e), mientras que su clave privada es d. Dicha clave pública se coloca en un directorio, al que cualquiera pueda acceder.

[1] La función de Euler $\phi(n)$ denota el número de enteros a tales que $1 \leq a \leq n$ y que son primos con n. Este número coincide con el número de elementos en el grupo multiplicativo $\mathbb{Z}^*_n$.

[2] La función de Euler es multiplicativa: dados dos números m y n coprimos, $\phi(m \cdot n) = \phi(m) \cdot \phi(n)$. Teniendo en cuenta que para un número primo n siempre se cumple que $\phi(n) = n - 1$, obtenemos la expresión anterior.

Normalmente, para un uso cotidiano, las claves RSA que se emplean hoy en día emplean tamaños de n de 2048 y 4096 bits, lo cual equivale a números con unas 616 y 1233 cifras decimales, respectivamente.

4.1.2. Cifrado de mensajes

Si un usuario A quiere mandar un mensaje a B, seguirá los siguientes pasos:

1. A obtiene la clave pública de B, (n_B, e_B), a través del directorio de claves.
2. A representa su mensaje m como un elemento de $\mathbb{Z}_{n_B}^*$ y primo con n_B.
3. Finalmente, calcula el criptograma $c = m^{e_A} \ (mod\ n_A)$ y lo envía a B.

4.1.3. Descifrado de mensajes

Cuando B reciba el criptograma, podrá descifrarlo aplicando la siguiente operación [RSA78]:

$$c^{d_B} = m^{e_B d_B} \ (mod\ n_B) \equiv m \ (mod\ n_B)$$

A continuación, veremos por qué dicha equivalencia es cierta. Obsérvese que $e_B \cdot d_B \ (mod\ n_B) \neq 1$, pues e_B y d_B son inversos en $\mathbb{Z}_{\phi(n_B)}^*$ y no en $\mathbb{Z}_{n_B}^*$. Para ello, debemos conocer dos importantes teoremas de la teoría de números: el Teorema de Euler y el pequeño Teorema de Fermat.

Teorema 4.1. ***(de Euler)*** *Para todo elemento $a \in \mathbb{Z}_n^*$ se verifica*

$$a^{\phi(n)} \equiv 1 \ (mod\ n)$$

Teorema 4.2. ***(pequeño de Fermat)*** *Si p es un número primo, para todo $a \in \mathbb{Z}_p$ se verifica:*

$$a^p \equiv a \ (mod\ p)$$

y si a no es divisible por p, entonces:

$$a^{p-1} \equiv 1 \ (mod\ p)$$

Vemos que, realmente, el pequeño teorema de Fermat es un caso específico del teorema de Euler, donde n es primo. Una vez vistos estos teoremas, podemos estudiar por qué el descifrado funciona en el algoritmo anterior.

Teorema 4.3. *Dada una clave pública (n, e), con $n = p \cdot q$ y una clave privada d, tales que $e \cdot d = 1 \ (mod\ \phi(n))$. Entonces, para todo entero $c = m^e \in \mathbb{Z}_n$ se cumple que:*

$$c^d \equiv (m^e)^d \ (mod\ n) \equiv m \ (mod\ n)$$

Demostración. Como sabemos que $e \cdot d \equiv 1 \ (mod\ \phi(n))$, entonces:

$$\exists k \in \mathbb{Z} \ : \ e \cdot d = 1 + k \cdot \phi(n)$$

Por otro lado, tenemos, por el teorema pequeño de Fermat, la siguiente congruencia:

$$m^p \equiv m \ (mod\ p)$$

Además, como $mcd(m, n) = 1$, podemos simplificar por m, obteniendo:

$$m^{p-1} \equiv 1 \ (mod\ p)$$

Como $p - 1$ divide a $\phi(n) = (p - 1)(q - 1)$, tenemos:

$$m^{ed} = m^{1+k\cdot\phi(n)} = m \cdot (m^{p-1})^{(q-1)k} \equiv m \cdot 1^{(q-1)k} \ (mod\ p) \equiv m \ (mod\ p)$$

De la misma forma, podemos demostrar que $m^{ed} \equiv m \ (mod\ q)$. Sabiendo que p y q son primos distintos, por el Teorema Chino del Resto [Bai19], llegamos a que $m^{ed} \equiv m \ (mod\ p \cdot q)$, y por tanto:

$$c^d \equiv (m^e)^d \ (mod\ n) \equiv m \ (mod\ n)$$

4.2. Seguridad de RSA

Para descifrar un criptograma cifrado con RSA, necesitamos calcular una clave de descifrado d a partir de los datos públicos n, e y C. Esto, a priori, es una tarea intratable si no se conoce el valor de $\phi(n)$. Sin embargo, un mal uso del criptosistema o una mala elección de claves/parámetros puede facilitar la ruptura del criptosistema, ya sea averiguando $\phi(n)$ o empleando otro tipo de ataques.

Para obtener $\phi(n)$, un atacante podría intentar descomponer n en sus factores primos p y q, y así obtener $\phi(n) = (p - 1)(q - 1)$, quedando el criptosistema roto. Es por esto que suele mencionarse que la seguridad de RSA recae en que no se conozca un algoritmo eficiente de factorización.

Actualmente, los mejores algoritmos de factorización de números requieren un tiempo subexponencial, por lo que la tarea se vuelve inabordable si n

es lo suficientemente grande. No obstante, existen algoritmos capaces de factorizar números que cumplan ciertas características, por lo que deberemos elegir cautelosamente p y q.

Además, también debemos ser precavidos eligiendo las claves e y d, pues una mala elección puede comprometer las claves. De la misma manera, hay que tener cuidado con qué mensaje M que se cifra y comprobar que no cumpla ciertas propiedades desfavorables.

4.2.1. Elección de p y q

Los números primos p y q que forman n deben seleccionarse de forma que dificulten lo máximo posible su factorización. A continuación, vamos a ver diferentes ataques realizables si no tomamos las precauciones necesarias.

p y q deben tener una longitud similar

Supongamos que usamos p y q tales que p sea mucho más pequeño que q. Entonces, un atacante podría, usando fuerza bruta, dividir n entre los todos los números primos que pueda, empezando por números pequeños. Si p es lo suficientemente pequeño, el atacante puede llegar a dividir entre dicho número y ver que el resto es 0, obteniendo como cociente q y por tanto factorizando n exitosamente.

Para que esto no suceda, debemos asegurarnos de elegir valores de p y q lo suficientemente grandes como para que no se pueda llegar a ninguno de los dos aplicando factorización por divisiones sucesivas.

p y q no deben ser demasiado cercanos

Tal y como es bueno evitar números bajos para p y q, también es bueno evitar que ambos números primos estén muy próximos. Si ese es el caso, entonces el atacante podría tantear y aplicar divisiones sucesivas con números cercanos a $\sqrt{n}$.

Otra opción, más eficiente, sería aplicar el *método de factorización de Fermat*, el cual veremos en el siguiente capítulo.

$mcd(p-1, q-1)$ debe ser pequeño

Proposición 4.1. *Dada una clave pública (n, e), un entero r sirve como exponente de descifrado si para todo mensaje M cumple las siguientes propiedades, equivalentes entre sí:*

1. $M^{er-1} \equiv 1 \ (mod \ n)$.
2. $M^{er-1} \equiv 1 \ (mod \ p)$, *y* $M^{er-1} \equiv 1 \ (mod \ q)$.
3. $e \cdot r - 1 \equiv 0 \ (mod \ p-1)$, *y* $e \cdot r - 1 \equiv 0 \ (mod \ q-1)$.
4. $e \cdot r \equiv 1 \ (mod \ mcm(p-1, q-1))$.

A raíz de la proposición anterior, otro posible ataque sale a la luz si $p-1$ y $q-1$ comparten un divisor grande. Tenemos que:

$$u = mcm(p-1, q-1) = \frac{\phi(n)}{mcd(p-1, q-1)}$$

Si $mcd(p-1, q-1)$ es grande, u es pequeño en comparación a $\phi(n)$. Entonces, si el atacante encuentra u y calcula d' tal que $e \cdot d' \equiv 1 \ (mod \ u)$, dicho d' funciona también como exponente de descifrado para todo mensaje m, de forma que el criptosistema queda roto.

$p-1$ y $q-1$ deben tener factores primos grandes

Otra medida de seguridad aplicable a los números p y q es hacer que ambos contengan factores primos grandes. Si no, $\phi(n) = (p-1)(q-1)$ tendría factores primos por debajo de una cota K, y esto permitiría a un atacante realizar un ataque de fuerza bruta para reconstruir $\phi(n)$ a partir de todas las posibles combinaciones de factores primos menores que K.

Es por esto, y por algunos ataques que veremos en el siguiente capítulo, que suele recomendarse que p y q sean *primos robustos*.

Definición 4.1. *Un número primo impar p es un primo robusto si:*

1. $p-1$ *tiene un factor primo grande* r.
2. $p+1$ *tiene un factor primo grande* s.
3. $r-1$ *tiene un factor primo grande* t.

4.2.2. Elección del exponente de cifrado

Normalmente, en aras del rendimiento, se suele emplear un exponente de cifrado pequeño. Así, cuando el usuario quiera cifrar un mensaje, al realizar la operación $c = m^e (mod \ n)$ tendrá que realizar menos multiplicaciones. Exponentes de cifrado comunes son $e = 3$ y $e = 2^{16} + 1$, cuyos cifrados implican 2 y 17 multiplicaciones usando el algoritmo de exponenciación rápida [Aca24], respectivamente.

El hecho de que varios usuarios compartan exponente de cifrado no supone ninguna amenaza, siempre y cuando todos ellos tengan distinto n (entonces, sus credenciales serían exactamente iguales, y podrían descifrar los mensajes de otros usuarios).

Una precaución que hay que tomar es que no se debe usar un e pequeño para cifrar un mismo mensaje m para varios destinatarios. Esto se debe a que un criptoanalista podría tomar los diferentes criptogramas resultantes y tratar de resolver el siguiente sistema de congruencias:

$$x \equiv c_1 \ (mod\ n_1), \quad x \equiv c_2 \ (mod\ n_2), \quad x \equiv c_3 \ (mod\ n_3)$$

que resulta ser resoluble empleando el Teorema Chino del Resto [Bai19].

Además, al usar exponentes de cifrado pequeños, debemos asegurarnos de que el mensaje es lo suficientemente grande como para que no sea posible calcularlo a partir de c usando la raíz del exponente de cifrado. Esto ocurre si $m^e < n$, y por tanto la operación módulo n no tiene efecto.

La solución más habitual a estos dos últimos problemas es el uso del *salado*[3] de mensajes. Esta técnica consiste en concatenar una cadena aleatoria al mensaje m antes de cifrarlo. Así, si queremos mandar un mismo mensaje m a varias personas, podemos concatenar a cada uno una cadena diferente, cifrando para cada uno mensajes diferentes. En el caso de los mensajes cortos, al salarlos resultan mensajes más largos, evitando que $m^e < n$ y, por tanto, que $m = \sqrt[e]{c}$.

4.2.3. Elección del exponente de descifrado

Para asegurar la seguridad de RSA, debemos asegurarnos de que el exponente de descifrado cumpla la siguiente propiedad:

$$longitud\ en\ bits(d) > \frac{1}{4}\ longitud\ en\ bits(n)$$

De no cumplirse que d sea lo suficientemente grande, existe un ataque capaz de calcular d de forma eficiente: el ataque de Wiener [Wie90]. Analizaremos este ataque en profundidad en el siguiente capítulo.

4.2.4. Propiedades del mensaje

n y el mensaje m deben ser coprimos

Si m y n no son primos entre sí, RSA queda roto ante el siguiente ataque:

[3] Del inglés, *salting*.

1. Si $mcd(m, p) \neq 1$, entonces $m = k \cdot p, k \in \mathbb{Z}$.
2. Si calculamos el criptograma como $c = m^e \ (mod\ n)$, tenemos:

   $$c = m^e + h \cdot n = p^e \cdot k^e + h \cdot p \cdot q = p(p^{e-1} \cdot k^e + h \cdot q) = p \cdot l$$

   y aplicando la operación módulo:

   $$c \equiv p \cdot l \ (mod\ n) \equiv p \cdot l \ (mod\ p \cdot q)$$

3. A no ser que $l = r \cdot q,\ r \in \mathbb{Z}$ (en cuyo caso, $c = 0$), calculando el máximo común divisor de n y c con el algoritmo de Euclides podemos conseguir p y factorizar n.

Por suerte, la probabilidad de elegir un mensaje m que sea múltiplo de p o q es despreciable, y se reduce de forma inversamente proporcional a la magnitud de los valores de p y q.

Mensajes inocultables

Los mensajes inocultables son aquellos que cumplen que, al aplicar el cifrado sobre ellos, el criptograma resultante es igual al mensaje original, es decir, $m^e \equiv m \ (mod\ n)$.

Dados p y q, podemos saber cuántos mensajes inocultables tiene el criptosistema empleando la siguiente fórmula:

$$(1 + mcd(e - 1, p - 1)) \cdot (1 + mcd(e - 1, q - 1))$$

Sabiendo que $e-1$, $p-1$ y $q-1$ son números pares, tenemos que habrán como mínimo nueve mensajes inocultables. Algunos de ellos pueden hallarse de forma trivial, como $m = 0$, $m = 1$ o $m = n - 1$.

Multiplicación de mensajes

Proposición 4.2. *El cifrado de RSA es homomórfico. Dada una clave pública* (n, e)*, dos mensajes* m_1 *y* m_2*, y* $C_1 \equiv m_1^e \ (mod\ n)$*,* $C_2 \equiv m_2^e \ (mod\ n)$*, siempre se cumple que* $(m_1 \cdot m_2)^e = m_1^e \cdot m_2^e \equiv C_1 \cdot C_2 \ (mod\ n)$*.*

Esta propiedad de RSA permite a un tercero realizar exitosamente un ataque adaptable al texto cifrado elegido. Si el adversario posee el criptograma $c = m^e \ (mod\ n)$, puede multiplicar el mismo por $k^e,\ k \in \mathbb{Z}$. Hecho esto, pide a un oráculo que descifre dicho mensaje, obteniendo el mensaje descifrado enmascarado:

$$\hat{m} \equiv (c \cdot k^e)^d \ (mod\, n) \equiv (m^e \cdot k^e)^d \ (mod\, n) \equiv (m \cdot k)^{ed} \ (mod\, n) \equiv m \cdot k \ (mod\, n)$$

Finalmente, el atacante puede calcular $m = \hat{m} \cdot k^{-1}$, descifrando el mensaje. Una estrategia similar puede emplearse para conseguir una firma digital con la clave d, simplemente intercambiando c por el mensaje m que se desee firmar.

Este ataque puede evitarse exigiendo ciertas propiedades estructurales a los mensajes en claro que se pueden cifrar.

Ataques cíclicos

Los ataques cíclicos se basan en la idea de que el cifrado repetido de un mismo mensaje en RSA lleva eventualmente al mismo mensaje. De esta manera, dado un criptograma c, esta estrategia busca un número k tal que:

$$c^{e^k} \equiv c \ (mod\ n)$$

Si llegamos al criptograma c tras realizar k cifrados, entonces el criptograma inmediatamente anterior, $c^{e^{k-1}}\ (mod\ n)$, debe ser congruente con $m\ (mod\ n)$, logrando así descifrar c. Claro que, para que este ataque tenga éxito, es posible que el usuario tenga que realizar hasta n cifrados, lo cual es inviable si el módulo RSA es lo suficientemente grande.

Por otro lado, un ataque que explota de forma más eficiente esta vulnerabilidad es el llamado *ataque cíclico generalizado*, cuyo objetivo es hallar el menor entero positivo k tal que:

$$u = mcd(c^{e^k} - c, n) > 1$$

Si encontramos k, entonces tendríamos que al menos una de las congruencias $c^{e^k} \equiv c\ (mod\ p)$, o $c^{e^k} \equiv c\ (mod\ q)$, es cierta. Esto llevaría automáticamente a que u es uno de los factores de n, logrando factorizar n.

4.3. Generación de números primos

Como hemos visto en las anteriores secciones, una de las tareas que debe realizar RSA es la generación de números primos de tamaños muy grandes. Además, se busca que estos primos cumplan ciertas propiedades para que el criptosistema sea seguro, como la robustez o que la diferencia entre p y q sea adecuada.

Una primera posibilidad que podemos contemplar es generar aleatoriamente un número r y comprobar a posteriori si es primo dividiéndolo entre los primos menores que $\sqrt{r}$. Sin embargo, aplicar divisiones sucesivas a números tan grandes es totalmente inviable.

En general, los algoritmos que determinan si un número es primo de forma absoluta tienen eficiencia subexponencial. No obstante, recientemente han surgido los primeros algoritmos polinómicos que comprueban la primalidad, como el test de primalidad AKS [AKS04]. A pesar de ser polinómicos, estos algoritmos suelen tener un orden de eficiencia muy alto, por lo que tampoco resultan útiles para este fin.

Por ello, haremos uso de los *test de primalidad probabilísticos*. Estos tests, en vez de comprobar si un número es primo con absoluta certeza, realizan ciertas pruebas que les permite determinar si es primo con cierta probabilidad. Dichas pruebas son propiedades matemáticas que cumplen todos los primos y *algunos* números compuestos, a los que denominaremos *pseudoprimos*.

4.3.1. Test de Fermat

El test de Fermat hace uso del *Teorema Pequeño de Fermat* para comprobar si un número es primo. El algoritmo consiste en elegir un número $1 < a < p - 1$ y realizar el siguiente cálculo:

$$a^{p-1} \equiv \begin{cases} 1 \ (mod\ p) & si\ p\ es\ primo \\ b \ (mod\ p),\ b \in \mathbb{Z}_p & si\ p\ es\ compuesto \end{cases}$$

Si bien un número primo siempre pasará el test independientemente del valor de a, un número compuesto podría pasarlo para ciertos valores de a "por casualidad". Para aumentar la fiabilidad del test, se pueden probar diferentes bases a con las que realizar la prueba. Si a es compuesto y no cumple la propiedad que comentamos en el siguiente párrafo, sabemos que fallará el test de Fermat por lo menos la mitad de las veces.

Desafortunadamente, existen algunos números compuestos b que cumplen la condición anterior una gran cantidad de bases a. Concretamente, cumplen que $a^b \equiv 1(mod\ p)$ si $a \in \mathbb{Z}_p^*$ Estos son los *números de Carmichael*, los cuales son muy difíciles de encontrar, pero su probabilidad no es despreciable. Por tanto, si encontramos uno de estos números, el Test de Fermat puede fallar incluso realizando muchas pruebas con diferentes bases. Por ejemplo, el primer número de Carmichael es el 561.

4.3.2. Test de Miller-Rabin

El test de Miller-Rabin mejora el test de Fermat con la noción de *pseudoprimo robusto*.

Definición 4.2. *Dado un número impar n tal que $n-1 = 2^s \cdot r$ con r impar, y $a \in [1, n-1]$, n es un pseudoprimo robusto en la base a si se cumple una de las siguientes condiciones:*

1. $a^r \equiv 1(mod\ p)$.
2. $\exists e : a^{2^e r} \equiv -1(mod\ p),\ con\ 0 < e < s$.

Mientras que todo primo cumple ser pseudoprimo robusto, dado un número compuesto c cualquiera, existen menos de $p/4$ bases a para las cuales c es pseudoprimo robusto. En consecuencia, cada vez que ejecutamos la prueba anterior con un número compuesto, la comprobación solo fallará un 25 % de las veces en el peor caso. En la práctica, el porcentaje de fallo suele ser bastante más bajo.

De forma similar al algoritmo de Fermat, podemos ejecutar la comprobación para diferentes valores de a. En dicho caso, la probabilidad de éxito del algoritmo viene dada por la fórmula $1 - \frac{1}{4^t}$, siendo t el número de pruebas realizadas. Con pocas iteraciones, es fácil conseguir un valor de confianza que haga despreciable la posibilidad de que un número sea falsamente identificado como un primo.

A diferencia del anterior algoritmo, no existe ningún número compuesto que sea *pseudoprimo robusto* para todas las bases posibles. Esto supone una clara ventaja de Miller-Rabin, pues no corre el peligro de equivocarse si realizamos las suficientes iteraciones.

La eficiencia del algoritmo de Miller-Rabin es $O(n^3)$.

Mejoras del Test de Miller-Rabin

En la práctica, podemos mejorar la eficiencia del test de Miller-Rabin empleando diferentes técnicas.

En primer lugar, podemos usar preliminarmente al test el algoritmo de las divisiones sucesivas con los primos menores que 256. Un resultado matemático nos indica que aproximadamente el 90 % de los números dividen a algún primo de los 54 primos por debajo de 256. Así, empleando dichas divisiones (que, como son con números que ocupan menos de un byte, son tremendamente rápidas) podremos ahorrarnos sobre el 90 % de las ejecuciones de la comprobación de Miller-Rabin, más costosa computacionalmente.

Otro truco para mejorar la eficiencia del test es emplear valores de a pequeños, en vez de elegirlos aleatoriamente en el rango $[1, n-1]$. Esto reduce la eficiencia teórica del algoritmo a $O(n^2)$, acelerando claramente el proceso.

4.3.3. Generación de primos robustos

En 1985, John Gordon propuso un algoritmo para generar números primos robustos para el criptosistema RSA [Gor85]. El algoritmo de Gordon funciona de la siguiente manera:

1. Generamos s y t, ambos primos de gran tamaño.

2. Elegimos aleatoriamente $i \in \mathbb{Z}$. A partir de él, calculamos $r = 2i \cdot t + 1$. Si este valor no es primo, aumentamos en 1 el valor de i y repetimos el proceso hasta obtener un r primo.

3. Calculamos $p_0 = 2s^{r-2} \ (mod\ r) \cdot s - 1$. Si p_0 no es impar, entonces volvemos al paso 1.

4. De forma similar al paso 2, elegimos aleatoriamente $j \in \mathbb{Z}$, y calculamos $p = p_0 + 2jrs$. Si este valor no es primo, incrementamos el valor de j en 1 hasta que lo sea. Una vez obtenido un p primo, p también será robusto.

Como resultado, tenemos que $s^{r-1} \equiv 1 \ (mod\ r)$ (por el teorema pequeño de Fermat). En consecuencia, $p_0 \equiv 1 \ (mod\ r)$ y $p_0 \equiv -1 \ (mod\ s)$. Sabiendo esto, es fácil ver que p es un primo robusto, pues:

$$p - 1 = p_0 + 2jrs - 1 \equiv 0 \ (mod\ r)$$

$$p + 1 = p_0 + 2jrs - 1 \equiv 0 \ (mod\ s)$$

$$r - 1 = 2it \equiv 0 \ (mod\ t)$$

Usando Miller-Rabin para generar los primos s, t y comprobar la primalidad de r y p, el tiempo de ejecución esperado de este algoritmo es únicamente un 19 % mayor que el de Miller-Rabin aplicado directamente a un primo del mismo tamaño que p. Claro que, si usamos test de primalidad probabilísticos para comprobar la validez de los primos, hemos de tener en cuenta que el algoritmo de Gordon será también probabilístico y su éxito dependerá del grado de confianza con que se comprueba la primalidad.

Otro factor importante a tener en cuenta es la elección de los valores aleatorios del algoritmo para que nos dé un número primo del tamaño que deseemos. Principalmente, el término que determinará el tamaño del primo será $2jrs$, pues sabemos que $p_0 < r \cdot s$. Así, si deseamos un tamaño de n para p, buscaremos que r y s tengan cada uno algo menos de $n/2$ bits. De esta forma, $2r \cdot s$ tendrá algo menos de n bits. Esos "algo menos" los determinaremos en la fase de implementación de manera aproximada. Para llegar a los n bits, emplearemos un tamaño de j que supla la diferencia de magnitud entre n y $log_2(2rs)$. La aproximación que usaremos en este trabajo será usar la diferencia de bits entre ambos para determinar el valor de j. De forma similar actuaremos para calcular el tamaño de t: usaremos un t con algo menos de $n/2$ bits, y supliremos la diferencia con la magnitud deseada de r con el valor de i.

Capítulo 5

Ataques al RSA en computación clásica

Como ya hemos mencionado, es esencial aplicar sobre los criptosistemas un criptoanálisis exhaustivo para asegurar su seguridad y, en caso de que se descubran debilidades en el mismo, tratar de solucionarlas. En este capítulo, estudiaremos los diferentes ataques y vulnerabilidades que se conocen del criptosistema RSA, así como las medidas necesarias para evitar su efectividad.

5.1. Algoritmos de factorización

En esta sección, vamos a centrarnos en aquellos ataques que tratan de obtener los primos p y q que componen n a partir de este último.

5.1.1. Métodos clásicos

Divisiones sucesivas

Este método de factorización se basa simplemente en probar a dividir n entre los primeros números primos, tal y como se factorizan los números en la escuela. De esta misma manera, podemos comprobar si el número n es primo, dividiendo por todos los primos t tales que $t \leq \sqrt{n}$ y comprobando que ninguna de las divisiones sea exacta.

Usando este método para la factorización, vemos entonces que el peor caso (cuando n es primo) es $O(|p : p \text{ } es \text{ } primo \text{ } y \text{ } p < \sqrt{n}|)$. Tomando el *Teorema de los Números Primos* [Wei24], sabemos que este número se aproxima a $\frac{n}{log\ n}$.

Con la tecnología actual, este algoritmo es factible solo para $n < 10^7$, por lo que no representa una amenaza para RSA por sí solo. Sin embargo, una ejecución parcial del mismo sí nos puede ser de utilidad en otros casos, como vimos en el apartado 4.3.2.

Método de Fermat

Este método es capaz de factorizar n si p y q son muy cercanos entre sí. Se basa en representar n como diferencia de cuadrados: $n = x^2 - y^2 = (x+y)(x-y) = p \cdot q$. Así, si encontramos x e y que cumplan dicha condición, habremos encontrado dos factores de n, que serían $p = x + y$ y $q = x - y$.

De esta forma, podemos plantear la siguiente ecuación:

$$x^2 - n = y^2$$

Si conseguimos encontrar un $x > \sqrt{n}$[1] tal que $x^2 - n$ dé como resultado un cuadrado perfecto, y^2, podremos calcular fácilmente $p = x+y$ y $q = x-y$, factorizando el número.

Si calculamos los valores de x e y en función de p y q, podemos ver que:

$$x = \left(\frac{p+q}{2}\right),\ y = \left(\frac{p-q}{2}\right)$$

Suponiendo que p y q son próximos entre sí, podemos esperar que $y = \frac{p-q}{2}$ sea un valor pequeño. Esto hará que, realizando un número de tanteos relativamente bajo, podamos encontrar un par (x, y) que satisfaga la ecuación.

Método de Kraitchik

En 1920, Maurice Kraitchik propone una mejora al algoritmo de Fermat [Rey18]. Esta consiste en buscar una diferencia de cuadrados, $x^2 - y^2$, divisible por n. Esto equivale a resolver la congruencia $x^2 \equiv y^2 \ (mod\ n)$. Cuando esta congruencia se cumple, sabemos que $(x + y)(x - y) = k \cdot n,\ k \in \mathbb{Z}$. Entonces, o bien $x + y$ o $x - y$ son divisores de n (cabiendo la posibilidad de que alguno de ellos sea n: en dicho caso, la solución no nos permite factorizar n), o bien algunos de los factores están en $x + y$ y otros están en $x - y$. Consecuentemente, si nos encontramos en el tercer caso, podemos conseguir factores no triviales de n aplicando el Algoritmo de Euclides sobre n y $x - y$.

Además, Kraitchik aportó una idea para agilizar la búsqueda de los x e y que solucionan la congruencia. Esta consiste en seleccionar valores de x_i

[1] Nótese que si $x \leq \sqrt{n}$, entonces $x^2 - n \leq 0$, por lo que no tendremos nunca un cuadrado perfecto en el otro lado de la ecuación.

cuyos factores primos sean pequeños y calcular $r_i = x_i^2 \ (mod\ n)$. Conforme vamos obteniendo los anteriores restos, comprobamos si mediante la multiplicación de varios de ellos podemos crear un cuadrado perfecto (lo cual, realmente, equivale a hacer que todos los factores de la multiplicación de restos tengan exponente par). Si ese es el caso, podemos obtener x e y, pues:

$$x^2 = (x_1 x_2 ... x_k)^2 \equiv r_1 r_2 ... r_k \ (mod\ n) \equiv y^2 \ (mod\ n)$$

5.1.2. Métodos modernos

Método ρ de Pollard

El método ρ (rho) de Pollard fue propuesto por Pollard en 1975, originalmente con el nombre de *Método de Monte Carlo para factorización*, debido a su naturaleza pseudoaleatoria [Pol75].

Con este método, creamos un conjunto finito C con n elementos. Concretamente, usaremos $\mathbb{Z}_n$. Sobre este conjunto, seleccionamos un elemento inicial $x_0 \in C$ y una función $f : C \longrightarrow C$ que tenga un comportamiento pseudoaleatorio (normalmente, se eligen polinomios de la forma $f(x) = x^2 + c$). Una vez tomados, creamos la secuencia $S = x_1, x_2, ...$ de forma recursiva como $x_i = f(x_{i-1})$. Así, obtendremos una secuencia que, llegado a un punto, formará un ciclo, dado que S no puede tomar infinitos valores diferentes.

De forma paralela, pensaremos en otra secuencia S', donde $x_i^{'} = x_i \ (mod\, p)$, siendo p un factor primo no trivial de n, inicialmente desconocido. Esta secuencia tendrá también un ciclo, debido a que se deriva directamente de S. Si encontramos una colisión en S', esto es, $x_i^{'} \equiv x_j^{'} (mod\ p)$, entonces tendremos que p divide a $x_i^{'} - x_j^{'}$. Por tanto, si calculamos $mcd(x_i^{'} - x_j^{'}, n)$ y el resultado es menor que n, habremos obtenido un factor no trivial de n.

Sin embargo, no conocemos p, por lo que no es viable calcular directamente S'. Por ello, el proceso anterior se lleva a cabo con S y verificando si $mcd(n, x_i - x_j) > 1$. En dicho caso, si $mcd(n, x_i - x_j) < n$, el resultado será p, factorizando finalmente n. Puede darse, rara vez, que la colisión también ocurra en S y $x_i = x_j$, de forma que $mcd(n, 0) = n$, fallando el método. En estos casos, se suele cambiar f o x_0 y volver a probar.

Dado que puede ser costoso almacenar S, una forma de encontrar colisiones es el algoritmo de búsqueda de ciclos de Floyd. Este usa dos variables: T y L, que iterativamente se calculan hasta lograr una colisión en S':

- $T_i = f(T_{i-1})$. Al avanzar despacio, podemos llamarla "tortuga".
- $L_i = f(f(L_{i-1}))$. Al avanzar más rápido, podemos llamarla "liebre".

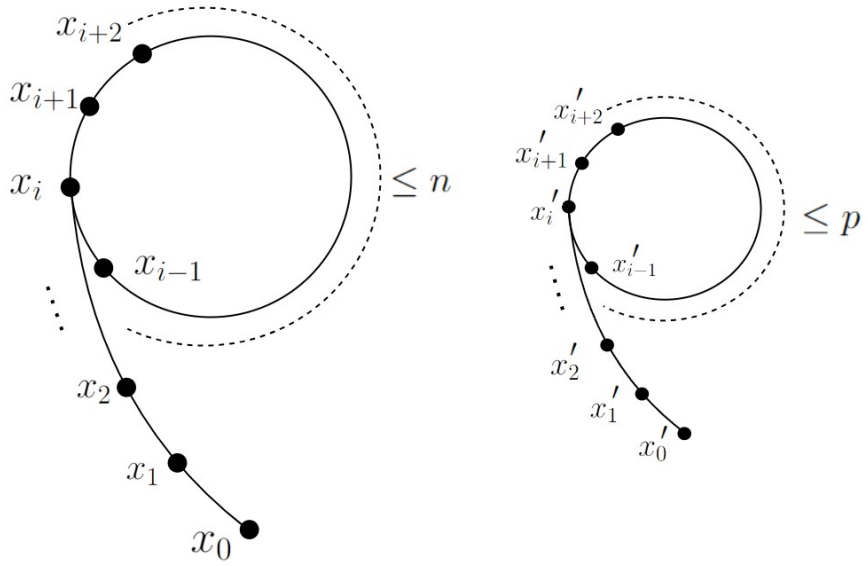

Figura 5.1: Secuencias S (a la izquierda) y S' (a la derecha) del método ρ de Pollard.

Como tanto T como L llegarán eventualmente a la parte cíclica de la secuencia, necesariamente llegará un momento en que ambas variables coincidan en el mismo punto de la misma, "alcanzando la liebre a la tortuga".

Aunque el ciclo de la secuencia S' puede llegar a ser de tamaño p, el tamaño esperado de este ciclo es $O(\sqrt{p})$. La *Paradoja del Cumpleaños* enuncia, precisamente, que si elegimos aleatoriamente números entre 0 y p, la cantidad de elecciones al azar (recordemos que f simula una función aleatoria) que tendremos que hacer hasta alcanzar una repetición es $\sqrt{p}$ (de media).

Esto hace que este algoritmo tenga que realizar, de media, $O(\sqrt{p}) \leq O(n^{\frac{1}{4}})$ multiplicaciones, aunque realmente sería $O(p)$ en el peor caso.

Método $p-1$ de Pollard

En 1974, Pollard describió en [Pol74] el método $\rho-1$ de Pollard: otro algoritmo de factorización de propósito especial, aunque con una idea completamente diferente al anterior. Este sirvió como inspiración a la factorización por curvas elípticas, que veremos más adelante [Ste09].

El método se basa en la suposición de que alguno de los factores primos del número a descomponer (por ejemplo, p), al restarle 1, sea una *B-potencia-uniforme*[2].

[2]Del inglés, "Power-Smooth".

Definición 5.1. *Dado un número n y su descomposición en números primos $p_1^{b_1} p_2^{b_2} ... p_n^{b_n}$, se dice que n es B-potencia-uniforme si todos sus factores cumplen que $p_i^{b_i} \leq B$.*

Sabiendo por el Teorema de Fermat que para cualquier $a > 1$ y p primo:

$$a^{p-1} \equiv 1 \ (mod\ p)$$

podemos calcular $m = mcm(1, 2, ..., B)$. Si $p - 1$ es *B-potencia-uniforme*, entonces todas las potencias de los factores primos que componen $p - 1$ estarán en el conjunto $1, 2, ..., B$, y por tanto $p - 1$ divide a m. Así que:

$$a^m = a^{(p-1)k} \equiv 1^k \ (mod\ p) \equiv 1 \ (mod\ p), \quad p|(a^m - 1)$$

Por tanto, p divide al $mcd(a^m - 1, N)$. Si $mcd(a^m - 1, N) < N$, entonces habremos conseguido un factor no trivial de N, factorizándolo.

También cabe la posibilidad de que $mcd(a^m - 1, N) = N$. Esto puede suceder si todos los factores de N menos 1 son *B-potencias-uniformes*. En este caso, podemos repetir el proceso disminuyendo el valor de B para evitar que todos los factores cumplan dicha propiedad, o evaluando $mcd(a^m - 1, N)$ en cada iteración. Otra opción es cambiar el valor a y volver a probar.

Por último, si la cota B resulta insuficiente y n no tiene primos tales que $p - 1$ sea *B-potencia-uniforme*, tendremos que $mcd(a^m - 1, N) = 1$.

Para calcular $mcm(1, 2, ..., B)$, nos bastará con seguir los siguientes pasos:

1. Inicializar una variable $m = 1$.
2. Para todo i desde 2 hasta B:
   a) Si i es primo, calculamos k tal que $i^k < B < i^{k+1}$ y acumulamos dicho valor con $m \longleftarrow m \cdot i^k$.
   b) Si i es compuesto, pasamos al siguiente número.

Método $p + 1$ de Williams

Este ataque fue propuesto por Williams en 1982 [Wil82]. En este caso, el algoritmo vulnera aquellos $n = pq$ tales que $p + 1$ tenga factores primos pequeños (en otras palabras, que sean *B-potencia-uniformes*). Su funcionamiento se basa en las llamadas *sucesiones de Lucas*.

Definición 5.2. *Dados $r, s \in \mathbb{Z}$ y α, β las raíces de $x^2 - rx + s$, se definen las sucesiones de Lucas como:*

$$U_n(r, s) = \frac{\alpha^n - \beta^n}{\alpha - \beta}$$

$$V_n(r,s) = \alpha^n + \beta^n$$

Esta sucesión se puede representar también de forma recursiva como:

$$U_0 = 0, U_1 = 1, U_n(r,s) = rU_{n-1}(r,s) - sU_{n-2}(r,s)$$

$$V_0 = 2, V_1 = r, V_n(r,s) = rV_{n-1}(r,s) - sV_{n-2}(r,s)$$

También se define el discriminante como $\Delta = (\alpha - \beta)^2 = r^2 - 4s$.

Esta sucesión se caracteriza por cumplir el siguiente teorema, que será el que nos ayude a definir el Algoritmo de Williams.

Definición 5.3. *Dado r un entero y p un primo impar, el símbolo de Legendre se define como*

$$\left(\frac{r}{p}\right) = \begin{cases} 0, & si\ p\ divide\ a\ r. \\ 1, & si\ r^2 \equiv 1\ (mod\ p) \\ -1, & si\ r^2 \equiv x\ (mod\ p),\ x \notin \{0,1\}. \end{cases}$$

Teorema 5.1. *Si p es un primo impar tal que $p \nmid s$ y el símbolo de Legendre $\left(\frac{\Delta}{p}\right) = \epsilon$, entonces para cualquier entero $m \geq 0$:*

$$U_{(p-\epsilon)m}(r,s) \equiv 0(mod\ p)$$

$$V_{(p-\epsilon)m}(r,s) \equiv 2s^{m(1-\epsilon)/2}(mod\ p)$$

Sea p un divisor de n, tal que $p+1$ sea *B-potencia-uniforme*, y $h = mcm(1..B)$, que sería el mayor número tal que sea *B-potencia-uniforme*. Podemos deducir, por tanto, que $(p+1)|h$. Aplicando el teorema anterior, si $mcd(s,n) = 1$ y $\left(\frac{\Delta}{p}\right) = -1$:

$$U_{(p+1)m}(r,s) \equiv 0(mod\ p)$$

$$V_{(p+1)m}(r,s) \equiv 2s^m(mod\ p)$$

se sigue que $p|U_h(r,s)$. Como p también divide a n, tenemos finalmente que $p|mcd(U_h(r,s),n)$.

La principal dificultad del algoritmo es que se debe cumplir que $\left(\frac{\Delta}{p}\right) = \epsilon = -1$ y no sabemos en principio el valor de p. Por definición, Δ debe ser un no-residuo cuadrático en $(mod\ p)$. El proceso para conseguir esto es complejo, y se detalla en [Wil82].

Una vez tenemos dicho problema solucionado, nos basta con calcular $mcd(n, U_h(r,s))$. Si el resultado es 1, n no tiene factores primos *B-potencia-uniformes*, por lo que tendríamos que aumentar el valor de la cota. Si el

resultado es n, o bien todos los factores son *B-potencia-uniformes*, o bien hemos tenido mala suerte eligiendo el valor inicial de la secuencia de Lucas. Si, finalmente, $1 < mcd(n, U_h(r,s)) < n$, habremos encontrado un factor de n y conseguido factorizarlo.

5.1.3. Métodos actuales

Factorización por curvas elípticas

Definición 5.4. *Sea $\mathbb{K}$ un cuerpo y a, $b \in \mathbb{K}$ tales que $\Delta_{a,b} = -16(4a^3 + 27b^2) \neq 0$, definimos la curva elíptica $E_{a,b}(\mathbb{K})$ de la siguiente manera:*

$$E_{a,b}(\mathbb{K}) = \{(x,y) \in \mathbb{K}^2 : y^2 = x^3 + ax + b\} \cup \{O\}$$

siendo O un objeto matemático fijo, que representa el punto del infinito.

La condición de que $\Delta_{a,b} = -16(4a^3 + 27b^2) \neq 0$ se debe a que, en otro caso, la curva tendría un punto singular, cortándose a sí misma o presentando picos, dejando de ser una curva elíptica.

Cuando tenemos una curva elíptica $E_{a,b}(\mathbb{K})$, podemos definir la suma de dos elementos de la misma de la siguiente manera:

1. Si alguno de los puntos es O, este actúa como elemento neutro de la suma, siendo $P_1 + O = P_1$.
2. Si tenemos $P_1 = (x,y)$ y $P_2 = (x,-y)$, entonces $P_1 + P_2 = O$. En este caso, diremos que P_1 y P_2 son opuestos, y su suma da el elemento neutro.
3. En otro caso, para $P_1 = (x_1, y_1)$ y $P_2 = (x_2, y_2)$ calculamos λ de la siguiente manera:
   - Si los dos puntos a sumar son el mismo, $\lambda = (3x^2 + a)/2y$, que corresponde a la pendiente de la recta tangente a la curva elíptica en dicho punto.
   - Si los puntos son diferentes, calculamos $\lambda = \frac{y_1 - y_2}{x_1 - x_2}$, siendo esta la pendiente de la recta que pasa por ambos puntos.
4. Calculamos la coordenada x_3 de la suma de P_1 y P_2 con $x_3 = \lambda^2 - x_1 - x_2$. Este será la coordenada x donde la recta que pasa por P_1 y P_2 corta de nuevo a la curva elíptica.
5. Finalmente, calculamos la coordenada y_3 de la suma de P_1 y P_2 como $y_3 = -\lambda x_3 - (y_1 - \lambda x_1)$. Geométricamente, P_3 es el opuesto a aquel punto de la curva elíptica que es cortado por la recta que pasa por P_1 y P_2.

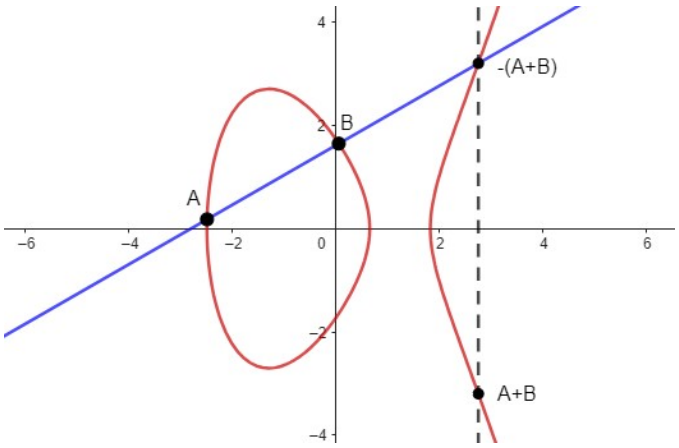

Figura 5.2: Ejemplo de suma en la curva elíptica $y^2 = x^3 + 2x - 1$ en $\mathbb{R}$ de los puntos A y B.

En la figura 5.2 podemos ver un ejemplo de suma en curvas elípticas de forma geométrica.

Una curva elíptica dotada con la operación suma anterior conforma un *grupo conmutativo*, esto es, la operación suma es conmutativa, asociativa, tiene elemento neutro y existe un inverso para cada punto.

Usando estas curvas, el matemático Hendrick Lenstra publica en 1987 un algoritmo de factorización inspirado en el método $p-1$ de Pollard [Len87]. Para entender este algoritmo, necesitamos conocer los siguientes teoremas:

Teorema 5.2. *Dada una curva elíptica $E_{a,b}(\mathbb{Z}_p)$ con p primo impar y un punto cualquiera Q no trivial en la curva, entonces $|E_{a,b}(\mathbb{Z}_p)|\cdot Q \equiv O \ (mod\ p)$, siendo $|E_{a,b}(\mathbb{Z}_p)|$ el número de elementos en la curva elíptica.*

Teorema 5.3. ***(de Hasse).*** *Dada una curva elíptica $E_{a,b}(\mathbb{Z}_p)$, el número de puntos pertenecientes a dicha curva está acotado y aleatoriamente distribuido (según los valores concretos de a y b) en el rango:*

$$[p + 1 - 2\sqrt{p}, p + 1 + 2\sqrt{p}]$$

El algoritmo consiste en lo siguiente:

1. Escogemos una pseudo-curva elíptica $E_{a,b}(\mathbb{Z}_n)$[3] y un punto $Q = (x, y) \in E_{a,b}(\mathbb{Z}_n)$, de forma que $y^2 \equiv x^3 + ax + b \ (mod\ N)$.

2. Calculamos L tal que, probablemente, sea múltiplo del número de elementos en la curva elíptica $E_{a,b}(\mathbb{Z}_p)$, con $p|n$. Podemos elegir, co-

[3]Nótese que, como $\mathbb{Z}_n$ no es un cuerpo, $E_{a,b}(\mathbb{Z}_n)$ no forma una curva elíptica. En estos casos, se les suele llamar a dichos objetos matemáticos pseudo-curvas elípticas.

mo en el método $p-1$ de Pollard, el mayor número tal que sea $k - potencia\ uniforme$.

3. Calculamos $L \cdot Q$. Este cálculo resulta eficiente si se realiza la multiplicación con un factor de L cada vez. En él, pueden surgir diferentes resultados:

   - Si mientras calculamos $L \cdot Q$ llegamos a cierta suma tal que $l \cdot Q = A + B = O$, entonces l divide al número de elementos que hay en n y, por ello, también divide al número de elementos de las curvas módulo los factores primos de n. En este caso, debemos bajar la cota k para que solo divida al orden de la curva elíptica de uno de los factores. Claro que, si $p = q$, ambos dividirán a n al mismo tiempo. Por ello, es conveniente comprobar si $\sqrt{n} \in \mathbb{Z}$, detectando dicho caso.
   - Si en el anterior caso llegamos a un $l \cdot Q = A + B$ tal que l divida al orden de solo una de las curva elípticas módulo sus factores (es decir, $l \cdot Q = O(mod\ p)$), cuando intentemos calcular la suma $(mod\ n)$ obtendremos que no podemos calcular la pendiente λ. Esto se debe a que, en dicho caso, la pendiente en la curva elíptica $E_{a,b}(\mathbb{Z}_p)$ es infinita, lo cual significa que se ha producido una división entre 0. Dicha división, si nos fijamos en el algoritmo de la suma, viene dada por $x_1 - x_2$, por lo que se cumple que $x_1 - x_2 \equiv 0\ (mod\ p)$. Cuando tratamos de realizar esta operación en $E_{a,b}(\mathbb{Z}_n)$, para calcular λ hay que obtener $(x_1 - x_2)^{-1}(mod\ n) = (x_1 - x_2)^{-1}(mod\ kp)$, el cual no existe debido a que $p|(x_1 - x_2)$. De hecho, si usamos el algoritmo de Euclides para calcular dicho inverso, obtendremos que $mcd(x_1 - x_2, n) = p$, factorizando n.
   - Finalmente, también es posible que consigamos calcular $L \cdot Q$ sin pasar por un $l \cdot Q = O$. En este caso, las curvas elípticas de los factores primos no son $k - potencias\ uniformes$. Para suplir este problema, podemos cambiar la curva elíptica alterando los valores a, b y repitiendo el proceso, con la esperanza de que el nuevo orden de la curva elíptica sí sea $k - potencia\ uniforme$. Esto es una ventaja respecto al algoritmo $p-1$ de Pollard, que falla siempre si $p-1$ presenta factores primos grandes.

Este algoritmo tiene un tiempo de ejecución esperado subexponencial. Concretamente, si tratamos de factorizar un módulo RSA $n = p \cdot q$, la eficiencia teórica es:

$$O(e^{(\frac{1}{2}+O(1))(ln\ n)^{\frac{1}{2}}(ln(ln\ n))^{\frac{1}{2}}})$$

Factorización mediante criba cuadrática

Como vimos en los métodos clásicos, podemos factorizar un número si conseguimos resolver la congruencia $x^2 \equiv y^2 \ (mod\ n)$, esto es, hallar un valor de x tal que al dividirlo por n nos dé como resto un cuadrado perfecto, resultando que $mcd(x-y,n) > 1$. Sin embargo, encontrar dicho valor de x mediante la fuerza bruta es inviable si n es grande.

La criba cuadrática es un método de factorización de propósito general propuesto por Carl Pomerance en 1982 (véase [Pom82] y [Pom96]) que busca reducir el tiempo de búsqueda de dicho x reutilizando la idea que tuvo Kraitchik: podemos calcular varios valores de y empleando la congruencia $x^2 \equiv r \ (mod\ n)$ y buscar un conjunto de restos $r_{0..k}$ tales que formen un cuadrado perfecto $y^2 = \prod_{i=0}^{k} r_i$.

Para buscar dicho conjunto de restos, podemos representar los diferentes r_i como vectores que codifican los exponentes de sus descomposiciones en factores primos. Por ejemplo, el número $84 = 2^2 \cdot 3 \cdot 7$ se representaría como $(2,1,0,1)$. De esta manera, el problema se reduce a encontrar una combinación lineal de vectores tal que todas las coordenadas sean pares o, lo que es lo mismo, 0 $(mod\ 2)$. Como solamente nos interesa la paridad de los exponentes, podemos directamente usar vectores con 0 si el exponente es par y 1 si el exponente es impar para el primo correspondiente.

La idea que propone la criba cuadrática es que si conseguimos $k+1$ restos tales que sean todos sus factores primos estén por debajo del primo número K (o, lo que es lo mismo, que sean *K-uniformes*), entonces podremos formar un sistema de $k+1$ vectores con k coordenadas, por lo que necesariamente debe existir una dependencia lineal entre dichos vectores, que podremos encontrar empleando, por ejemplo, el *método de Gauss*. Equivalentemente, habremos encontrado el producto de restos necesario para encontrar nuestro y^2, potencialmente factorizando n.

Para la elección de K, debemos tener en cuenta que si escogemos un valor demasiado pequeño, entonces es posible que no existan $k+1$ restos que nos permitan encontrar un resto cuadrático, mientras que un valor mayor de K hará que nuestros números sean más grandes y necesitemos más restos cuadráticos para llegar a la solución. Por ello, la virtud está en el punto medio, donde tengamos suficientes restos y estos no sean muy grandes.

Para encontrar restos r_i que sean *K-uniformes*, el algoritmo realiza una criba empleando la siguiente expresión.

$$f(x) = x^2 - n$$
$$f(x+kp) = (x+kp)^2 - n = x^2 + 2xkp + (kp)^2 - n$$
$$= f(x) + 2xkp + (kp)^2 \equiv f(x) \ (mod\ p)$$

Por tanto, si encontramos una solución para $f(x) \equiv 0 \ (mod\ p)$, habremos generado una secuencia de números que cumplen ser divisibles por p. Realizando los cálculos oportunos, obtenemos que la congruencia anterior tendrá dos soluciones si se cumple que $n \equiv y^2 \ (mod\ p)$, con $0 < y < p$, y ninguna en otro caso. El hecho de que esta congruencia no tenga soluciones para ciertos valores de p hace que sean menos los primos que tengamos que considerar en los vectores que representan los restos. A estos valores de p por debajo del k-ésimo primo que tienen soluciones para la congruencia anterior los llamaremos la *base de primos*.

Una vez tenemos nuestra base de primos, calculamos cierto número de restos resultantes de $f(x)$ para $x > \sqrt{n}$ (así solo trabajamos con restos positivos, aunque en [Pom96] se menciona la posibilidad de incluir restos negativos). Estos formarán nuestra tabla sobre la cual aplicaremos la criba. Después, para cada primo en la base de primos p, calculamos las soluciones de x en $f(x) = x^2 - n \equiv 0 \ (mod\ p)$. Despejando, tenemos que $x \equiv \sqrt{n} \ (mod\ p)$. Como $f(x + kp) \equiv f(x) \ (mod\ p)$, si $f(a) \equiv 0 \ (mod\ p)$, toda $f(a + kp), k \in \mathbb{Z}$ será divisible por p. Entonces, sobre la tabla que hemos creado, dividimos todos los $f(a + kp)$ por p.

Cuando hayamos recorrido toda nuestra base de primos, podemos detectar aquellos $f(x)$ que son *K-uniformes* observando si su valor tras aplicar la criba es 1. Entonces, todos los factores primos del número estarán en la base de primos, y podremos representarlo con la notación vectorial que hemos definido antes.

Después de que detectemos los suficientes restos *K-uniformes*, podremos obtener un resto cuadrático mediante una combinación lineal de los mismos. Finalmente, cuando encontremos dicha combinación lineal para formar el cuadrado perfecto, factorizamos como lo haríamos en el método de Fermat.

Cabe destacar que este algoritmo solo resulta apropiado si el número a factorizar es muy grande, siendo más eficiente emplear los enfoques anteriores en otro caso.

Factorización mediante criba general del cuerpo de números

Asintóticamente, la criba general del cuerpo de números es el algoritmo más eficiente a día de hoy para factorizar un número [Pom96]. Este algoritmo se basa en la criba cuadrática de Pomerance, pero hace uso de *cuerpos numéricos* para buscar números candidatos a ser uniformes. Este algoritmo es bastante complejo, lo cual hace que solo merezca la pena usarlo para números de más de 130 dígitos decimales. La eficiencia teórica del algoritmo es:

$$O(e^{\left(\sqrt[3]{\frac{64}{9}}+O(1)\right)(ln\ n)^{\frac{1}{3}}(ln\ ln\ n)^{\frac{2}{3}}})$$

5.2. Ataques especializados al RSA

En esta sección, vamos a explorar un ataque al RSA que evita enfrentarse contra la factorización de p y q, que suele ser inviable para los tamaños y algoritmos de factorización actuales.

5.2.1. Ataques basados en fracciones continuas

En 1990, Wiener propuso un algoritmo eficiente que permite determinar un número racional $r = \frac{a}{b}$ a partir de una aproximación suya r' suficientemente buena [Wie90]. Este algoritmo está basado en el concepto de *fracción continua*.

Fracciones continuas

Definición 5.5. *Las fracciones continuas de una sucesión de números* $S = \{s_0, s_1, s_2, ...\}$ *consiste en otra sucesión definida como:*

$$\langle s_0, s_1, s_2, ...\rangle = s_0, s_0 + \frac{1}{s_1}, s_0 + \frac{1}{s_1 + \frac{1}{s_2}}, s_0 + \frac{1}{s_1 + \frac{1}{s_2 + \frac{1}{s_3}}}, ...$$

A cada elemento i-ésimo de una fracción continua se le denomina reducida i-ésima. Si los diferentes s_i, denominados cocientes incompletos, son números naturales distintos de cero, entonces tenemos una fracción continua ordinaria.

Teorema 5.4. ***(de las fracciones continuas)***. *Todo número racional se puede desarrollar en fracción continua ordinaria. Si el último cociente incompleto es 1, este desarrollo es único, y se puede calcular con el algoritmo de las fracciones continuas.*

El algoritmo de las fracciones continuas funciona de una manera muy similar al Algoritmo de Euclides. Si tenemos la fracción a/b, podemos expresarla como fracción continua $\langle a_0, a_1, ..., a_i\rangle$, donde cada a_i es el cociente obtenido en la división i-ésima del Algoritmo de Euclides.

A partir de estas premisas, Wiener diseñó un algoritmo que utiliza como entrada la expresión en fracción continua de r', siendo $r' \approx r$.

- Para $i = 1..n$, siendo n el número de cocientes incompletos que tiene la expresión en fracción continua de r':
    1. Calculamos el numerador y denominador de la reducida i-ésima de r': $R_i = p_i/q_i$. Para ello, simplemente realizamos las diferentes sumas y divisiones en la fracción continua hasta llegar a una fracción.

2. Calculamos el desarrollo en fracción continua de $R_i = \langle s'_0, s'_1, ..., s'_n \rangle$ usando el algoritmo de las fracciones continuas.
3. Calculamos $\tilde{r}_i$, determinado por la expresión:

$$\tilde{r}_i = \begin{cases} \langle s'_0, s'_1, ..., s'_n + 1 \rangle & si\ i\ es\ par \\ \langle s'_0, s'_1, ..., s'_n \rangle & si\ i\ es\ impar \end{cases}$$

4. Comprobamos si $\tilde{r}_i = r$ mediante cualquier método. Si es el caso, terminamos la ejecución.

Teorema 5.5. *Si tenemos que $r = \frac{p}{q}$ y la diferencia $\delta = 1 - \frac{r'}{r}$ cumple que $\delta < \frac{2}{3pq}$, entonces el algoritmo de Wiener tendrá éxito. No obstante, si no se cumple esta condición, aún puede darse el caso de que funcione.*

Ataque de Wiener

Debido a la construcción de RSA, sabemos que:

$$e \cdot d = 1 + k \cdot \phi(n) \equiv 1\ (mod\ \phi(n))$$

Empleando la anterior fórmula, tenemos que:

$$\frac{e}{\phi(n)} - \frac{k}{d} = \frac{1}{d\phi(n)}$$

Como d y $\phi(n)$ son potencialmente números muy grandes, el término $\frac{1}{d\phi(n)}$ tenderá a ser muy pequeño. Además, podemos considerar que n es razonablemente parecido a $\phi(n)$, pues $\phi(n) = (p-1)(q-1) = pq-p-q+1 = n-p-q+1$. Como p y q serán números muy grandes, su producto n será mucho más grande que ellos por separado. De esta manera, establecemos la siguiente aproximación:

$$\frac{k}{d} \approx \frac{e}{n}$$

El ataque de Wiener trata de hallar el exponente de descifrado d usando este $\frac{e}{n}$ como una aproximación al valor de la fracción $\frac{k}{d}$, para lo cual emplearemos el algoritmo de Wiener.

Sin embargo, debemos definir alguna forma de comprobar en cada iteración del algoritmo si el valor de $\tilde{r}_i = a/b$ es igual a nuestro k/d. Con el fin de detectarlo, seguiremos los siguientes pasos:

- Si $ed \equiv 1\ (mod\ \phi(n))$, como $\phi(n)$ es un número par, entonces $e \cdot d$ es impar. En consecuencia, e y d siempre serán números impares, por lo que podemos ignorar aquellos $\tilde{r}_i$ con denominador impar.

- Calculamos $C = \frac{e \cdot b - 1}{a}$. Si $a/b = k/d$, entonces $C = \frac{e \cdot d - 1}{k} = \phi(n)$. Si $C \notin \mathbb{Z}$, entonces podemos descartarlo.

- Si los dos puntos anteriores se cumplen, entonces probamos si se cumple la ecuación $e \cdot d = 1 + k \cdot \phi(n)$. Si es el caso, obtenemos el exponente de descifrado d y rompemos el criptosistema.

- Además, si nos interesa factorizar el módulo de RSA n, podemos formular la ecuación $(x - p)(x - q) = x^2 - (p + q)x + n$. Como $\phi(n) = (p-1)(q-1) = n-(p+q)+1$, y entonces $p+q = n-\phi(n)+1$, si sustituimos $p + q$ por esta nueva expresión en la anterior fórmula, llegamos a:

  $$(x - p)(x - q) = x^2 - (n - \phi(n) + 1)x + n$$

  De forma que, si resolvemos $x^2 - (n - C + 1)x + n = 0$ y $C = \phi(n)$, las soluciones serán precisamente p y q, factorizando n.

Este método de factorización, en términos de parámetros de RSA, funcionará siempre si $q < p < 2q$ y $d \leq \frac{1}{3}\sqrt[4]{n}$. En otro caso, no podemos asegurar poder romper el criptosistema, pero sigue siendo posible que el ataque tenga éxito.

Capítulo 6

Ataques cuánticos al RSA

La computación cuántica plantea una clara amenaza al criptosistema RSA. Esto se debe a la existencia de un algoritmo cuántico capaz de resolver el problema de la factorización de un número empleando un número de operaciones polinómico. Este es el llamado *algoritmo de Shor*, descrito por Peter Shor en 1999 en el artículo [Sho99]. En este capítulo, estudiaremos en profundidad este algoritmo, basándonos en el libro [NC10] y en el artículo original.

6.1. Algoritmo de Shor

Definición 6.1. *Un número $X \in \mathbb{Z}_n$ es una* ***raíz cuadrada no trivial de la unidad módulo*** *n si, siendo $1 < x < n-1$, se cumple que:*

$$x^2 \equiv 1 \ (mod\ n)$$

El algoritmo de Shor se basa en la idea de encontrar dicha raíz cuadrada no trivial de la unidad módulo n, donde n es nuestro número a factorizar. De esta manera, tendremos que $x^2 \equiv 1 \ (mod\ n)$ y, despejando, $x^2 - 1 = (x+1)(x-1) \equiv 0 \ (mod\ n)$, pudiendo factorizar n aplicando el máximo común divisor de n con cualquiera de los dos factores.

Definición 6.2. *El* ***orden de un número*** *a* ***módulo*** *n es el menor número $r > 0$ tal que:*

$$a^r \equiv 1 \ (mod\ n)$$

La fortaleza del algoritmo de Shor se basa en que existe un algoritmo cuántico eficiente capaz de calcular el periodo de una función periódica, como es $f(x) = a^x \ (mod\, n)$, con $1 < a < n-1$. De esta manera, podemos calcular el orden de a módulo n. Si dicho orden es par, entonces $a^r = (a^{\frac{r}{2}})^2 \equiv 1 \ (mod\, n)$, por lo que $a^{\frac{r}{2}}$ sería una raíz cuadrada no trivial módulo n.

Teorema 6.1. *Sea $n \in \mathbb{N}$ un número compuesto con m factores primos diferentes, $y \in \mathbb{N}$ un número aleatorio en el rango $[2, n-2]$, y r el orden de dicho número entero módulo n. Entonces, la probabilidad de que r sea par e $y^{r/2}$ $(mod\ n)$ sea una raíz cuadrada no trivial (esto es, no es ni 1 ni $n-1$) es mayor que $1 - (\frac{1}{2})^m$.*

El teorema anterior nos asegura que, con pocos intentos, podemos encontrar un a que produzca la raíz cuadrada no trivial que buscamos, resolviendo el problema. Por tanto, la parte clásica del algoritmo de Shor consta de los siguientes pasos:

1. Elegimos un número a aleatorio.
2. Calculamos el $mcd(a, n)$. Si no es 1, a es un factor no trivial de n y terminamos. En otro caso, seguimos con el siguiente paso.
3. Usamos el algoritmo cuántico para calcular el periodo de la función $f(x) = a^x\ (mod\ n)$, que coincide con el orden r de a módulo n.
4. Si r es impar o la raíz cuadrada encontrada es $n-1$ (una solución trivial), volvemos al paso 1.
5. Encontramos los factores de N con el $mcd(a^{r/2}+1, n)$ o $mcd(a^{r/2}-1, n)$, terminando el algoritmo.

Por ejemplo, si queremos factorizar el número $n = 15$, elegimos un número $a = 2$ que es coprimo con n y buscamos el periodo de la función $f(x) = a^x\ (mod\ 15)$. Como $2^4 = 16 \equiv 1\ (mod\ 15)$, el orden de 2 módulo n es 4. Como este orden es par, podemos calcular una raíz cuadrada no trivial de n como $2^{4/2}\ (mod\ 15) = 4$. En efecto, $4^2 \equiv 1\ (mod\ 15)$. Finalmente, se consigue la factorización con $mcd(4-1, 15) = 3$ y $mcd(4+1, 15) = 5$.

6.1.1. Búsqueda del periodo de una función

En este apartado, vamos a explorar cómo un circuito cuántico puede resolver el problema de encontrar el periodo de una función con un número de operaciones polinómico.

Transformada Cuántica de Fourier (QFT)

Para construir el algoritmo anterior, necesitaremos aplicar un subcircuito denominado *Transformada Cuántica de Fourier*, o *QFT* para abreviar. Tal

y como la transformada de Fourier discreta clásica, este circuito transforma los qubits empleando la siguiente expresión:

$$QFT\,|x\rangle = \frac{1}{2^{n/2}} \sum_{y=0}^{2^n-1} e^{2\pi i \frac{xy}{2^n}}\,|y\rangle$$

donde n es el número de qubits total del estado $|x\rangle$.

Nota: Cabe mencionar que un estado con varios qubits $|\psi\rangle = |x_i...x_0\rangle$ puede representarse como $|X\rangle$, donde X es el número decimal correspondiente al número binario $x_i...x_0$. Así, por ejemplo, el estado $|110\rangle$ puede abreviarse como $|6\rangle$.

Otra forma alternativa de calcular la QFT, la cual suele resultar muy práctica debido a que calcula el estado de cada qubit de manera independiente, es la siguiente:

$$|X\rangle \rightarrow \frac{(|0\rangle + e^{2\pi i \frac{x_0}{2}}\,|1\rangle) \otimes (|0\rangle + e^{2\pi i (\frac{x_1}{2} + \frac{x_0}{2^2})}\,|1\rangle) \otimes ... \otimes (|0\rangle + e^{2\pi i (\frac{x_{n-1}}{2} + ... + \frac{x_0}{2^n})}\,|1\rangle)}{2^{n/2}}$$

La aplicación de la QFT sobre un estado $|X\rangle = |x_{n-1}...x_1x_0\rangle$ donde cada qubit $|x_i\rangle$ es $|0\rangle$ o $|1\rangle$ produce una serie de qubits en estado de superposición, situados en el plano XY de la esfera de Bloch, con diferentes fases. Partiendo del estado de Bell $|+\rangle$, cada qubit x_j gira en la esfera de Bloch sobre el eje Z en sentido antihorario (es decir, cambia de fase) un ángulo $\alpha = \frac{X2\pi j}{2^n}$.

Una analogía que puede ser útil para ganar una intuición más clara del funcionamiento de la QFT es pensar que los diferentes qubits resultantes son "relojes"que van girando a diferentes velocidades conforme aumenta el valor de X. Concretamente, cada qubit gira la mitad de rápido que el anterior. De esta manera, el qubit $|x_{n-1}\rangle$ da una vuelta completa cada dos incrementos de X, el siguiente cada cuatro, el $|x_1\rangle$ la completará cada 2^{n-1} incrementos, y el $|x_0\rangle$ cada 2^n, es decir, al recorrer todos los estados básicos representables con n qubits.

Veamos un ejemplo de la aplicación de la QFT sobre el estado $|3\rangle = |1\rangle \otimes |1\rangle$:

$$QFT\,|3\rangle = \frac{1}{2}\sum_{y=0}^{3} e^{2\pi i \frac{3y}{4}}\,|y\rangle$$

$$= \frac{1}{2}(e^{2\pi i \frac{3\cdot 0}{4}}\,|0\rangle + e^{2\pi i \frac{3\cdot 1}{4}}\,|1\rangle + e^{2\pi i \frac{3\cdot 2}{4}}\,|2\rangle + e^{2\pi i \frac{3\cdot 3}{4}}\,|3\rangle)$$

$$= \frac{1}{2}(e^{0}\,|0\rangle + e^{\frac{3}{2}\pi i}\,|1\rangle + e^{3\pi i}\,|2\rangle + e^{\frac{9}{2}\pi i}\,|3\rangle)$$

$$= \frac{1}{2}(e^{0}\,|00\rangle + e^{\frac{3}{2}\pi i}\,|01\rangle + e^{3\pi i}\,|10\rangle + e^{\frac{9}{2}\pi i}\,|11\rangle)$$

$$= \frac{(|0\rangle + e^{3\pi i}|1\rangle) \otimes (|0\rangle + e^{\frac{3}{2}\pi i}|1\rangle)}{2} = ...$$

Si tenemos en cuenta que dar un giro de 2π es equivalente a dar una vuelta completa, podemos recalcular los ángulos para que no sean superiores a 2π, quedándonos con:

$$... = \frac{(|0\rangle + e^{\pi i}|1\rangle) \otimes (|0\rangle + e^{\frac{3}{2}\pi i}|1\rangle)}{2} = \frac{(|0\rangle + e^{2\pi i\frac{1}{2}}|1\rangle) \otimes (|0\rangle + e^{2\pi i(\frac{1}{2}+\frac{1}{4})}|1\rangle)}{2}$$

Que es la misma expresión que hubiéramos obtenido aplicando la fórmula alternativa.

Antes de poder crear el circuito para calcular la QFT, debemos presentar una nueva puerta que usaremos en el mismo. Esta puerta es la puerta de cambio de fase, que dado un ángulo α, gira el qubit sobre el eje Z un ángulo α. La representaremos con el símbolo R_α, siendo α el ángulo concreto que se gira el qubit.

$$R_\alpha = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\alpha} \end{bmatrix} \tag{6.1}$$

También nos será de utilidad definir una puerta que intercambie dos qubits de posición. Esta es la puerta *swap*, representada por una línea con una cruz en las líneas correspondientes a los qubits a intercambiar.

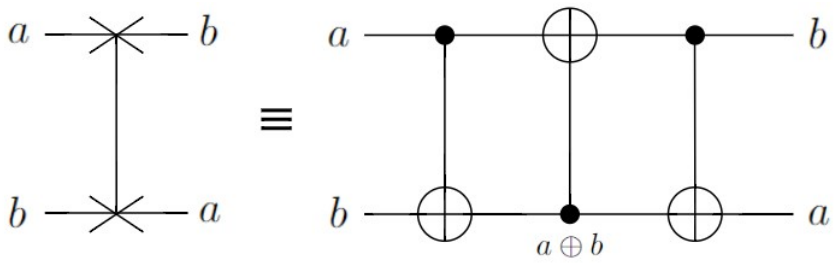

Figura 6.1: A la izquierda, representación de la puerta SWAP. A la derecha, una posible implementación de dicha puerta empleando puertas CNOT.

Una vez provistos de las puertas necesarias, vamos a implementar el circuito que calcula la QFT. Nos basaremos en la fórmula alternativa, pues nos permite calcular de uno en uno los qubits resultantes.

El primer qubit que debemos calcular es $|0\rangle + e^{2\pi i(\frac{x_{n-1}}{2}+...+\frac{x_1}{2^{n-1}}+\frac{x_0}{2^n})}|1\rangle$, pues es el único que en su cálculo intervienen todos los qubits de entrada. Este cálculo lo haremos sobre el qubit x_{n-1}, ya que no lo necesitaremos para el cálculo de los demás qubits de la QFT. Así, sobre x_{n-1} vamos a usar una puerta H, que mandará nuestro qubit al estado $|0\rangle + e^{2\pi i(\frac{x_{n-1}}{2})}|1\rangle$. Posteriormente, una puerta $R_{\pi/2}$ con control en x_{n-2} girará el estado un

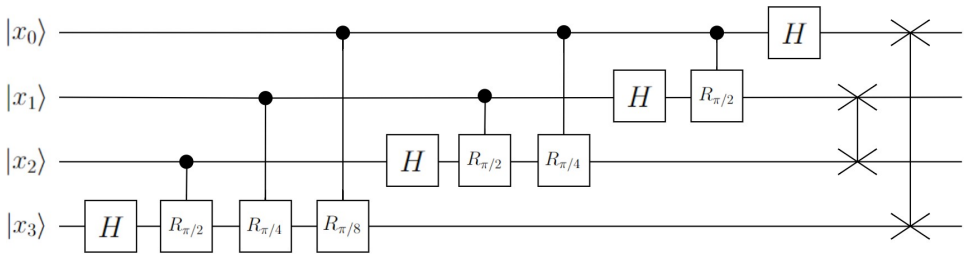

Figura 6.2: Circuito de la QFT para 4 qubits.

ángulo de 90 si x_{n-2} está activado, y así sucesivamente con todos los giros controlados a ejecutar.

Los siguientes qubits se calculan de manera similar, pero usando únicamente los qubits no usados hasta el momento. Finalmente, nos encontramos con que hemos calculado todos los qubits de la salida, pero están ordenados de manera inversa. Por ello, empleamos las puertas SWAP necesarias para que la salida sea la esperada.

Empleando el enfoque anterior para el cálculo de la QFT, el número de puertas usadas es $\frac{n(n+1)}{2}$, que es $O(n^2)$. Sin embargo, existen otras técnicas que permiten mejorar esta eficiencia a $O(n \cdot log\ n)$ [HH00].

En la figura 6.2, podemos ver un ejemplo concreto de circuito que calcula la transformada cuántica de Fourier para una entrada de 4 qubits.

Funciones en computación cuántica

En computación cuántica, debido a la deseada reversibilidad de los circuitos, las funciones se suelen definir de la siguiente manera:

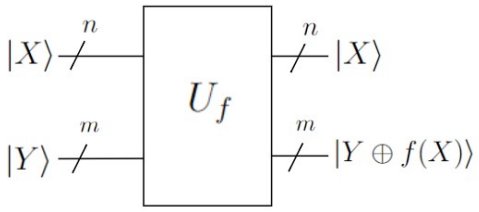

Figura 6.3: Representación de una función en un circuito cuántico.

Como vemos en la figura 6.3, la función recibe como entradas tanto los qubits X sobre los que se calcula la función $f(X)$, como los qubits donde escribir la salida Y. Dicha salida se escribe empleando la operación $\oplus$ sobre Y. De esta manera, puede deshacerse fácilmente la ejecución de la función volviendo a aplicar la misma sobre su propia salida.

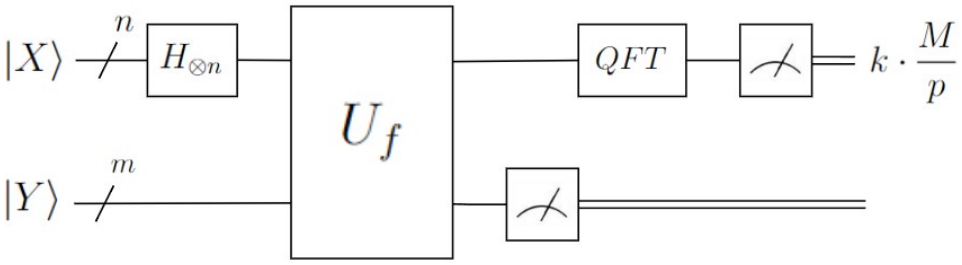

Figura 6.4: Circuito para el cálculo del periodo de la función U_f.

Cálculo del periodo de una función

Para realizar una explicación más digerible del funcionamiento del algoritmo de búsqueda del orden, distinguiremos dos casos, de forma similar a como se expone en [Mor03]. En primer lugar, trataremos sobre cómo encontrar el periodo p de la función en el caso especial donde p es una potencia de 2. Posteriormente, hablaremos del caso general, con p sin restricciones.

Periodos potencias de 2

Para encontrar el periodo p de una función de la forma $f(x) = a^x \ (mod\, n)$, crearemos un circuito U_f que implemente dicha operación en un circuito cuántico reversible. Este circuito tendrá un número $\epsilon = \lceil log_2(n) \rceil$ de qubits de entrada, y otros $s = \lceil log_2(n) \rceil$ qubits de salida. En la entrada, por tanto, podremos representar $M = 2^\epsilon$ entradas "clásicas"diferentes.

Una vez construida U_f, podemos hallar su periodo p a través del circuito cuántico en la figura 6.4.

Analicemos paso a paso qué hace este circuito.

1. Comenzamos con todos los qubits inicializados con el valor $|0\rangle$.

$$|\psi_0\rangle = |0\rangle^{\otimes(\epsilon+s)}$$

2. Aplicamos puertas de Hadamard sobre cada qubit en X, creando un estado de superposición en la entrada a la función.

$$|\psi_1\rangle = \frac{1}{\sqrt{M}} \sum_{x=0}^{M-1} |x\rangle\, |0\rangle^{\otimes s}$$

3. El estado anterior entra en la función U_f, dándonos como salida una

superposición de todas las evaluaciones de la función posibles.

$$|\psi_2\rangle = \frac{1}{\sqrt{M}} \sum_{x=0}^{M-1} |x\rangle\,|0 \oplus f(x)\rangle = \frac{1}{\sqrt{M}} \sum_{x=0}^{M-1} |x\rangle\,|f(x)\rangle$$

4. En este punto, medimos los valores de los m últimos qubits, correspondientes a la salida de la función. Esto causa que el estado colapse a una evaluación $f(x)$ específica, quedándonos en la parte de la entrada únicamente la posibilidad de medir aquellos x tales que $f(x) = \textit{medición} = x_0$. Como f es periódica, si su periodo es p y es potencia de 2 (y, por tanto, $p \mid M$), el número de resultados de x posibles es $r = \frac{M}{p}$.

$$|\psi_3\rangle = \sqrt{\frac{p}{M}} \sum_{k=0}^{r-1} |x_0 + k \cdot p\rangle$$

Este resultado genera, por tanto, una superposición de estados donde de forma periódica tenemos un estado elemental con amplitud $\sqrt{\frac{p}{M}}$, teniendo el resto de estados amplitud 0.

5. Cuando aplicamos la transformada cuántica de Fourier inversa sobre el anterior estado, obtenemos un nuevo estado. Como el estado anterior es periódico, su transformada de Fourier también será periódica, con periodo proporcional a $1/p$:

$$|\psi_4\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} \left[\frac{1}{\sqrt{M}} \sum_{y=0}^{M-1} e^{-2\pi i \frac{(x_0+kp)y}{M}} |y\rangle \right] =$$

y, cambiando el orden de las sumatorias,

$$= \frac{1}{\sqrt{p}} \sum_{y=0}^{M-1} \left[\frac{1}{r} \sum_{k=0}^{r-1} e^{-2\pi i \frac{kp \cdot y}{M}} \right] \cdot e^{-2\pi i \frac{x_0 \cdot y}{M}} |y\rangle$$

En este resultado, podemos observar que $\frac{1}{r} \sum_{k=0}^{r-1} e^{-2\pi i \frac{kp \cdot y}{M}}$ puede valer 0 o 1, dependiendo especialmente del valor de y. Si $y = tr$ con $t \in \mathbb{N}$, esto es, si y es múltiplo de r:

$$\frac{1}{r} \sum_{k=0}^{r-1} e^{-2\pi i \frac{kp \cdot y}{M}} = \frac{1}{r} \sum_{k=0}^{r-1} e^{-2\pi i \frac{kp \cdot tr}{M}} = \frac{1}{r} \sum_{k=0}^{r-1} e^{-2\pi i k \cdot tr}$$

Como $-2\pi i k \cdot tr$ siempre es un número múltiplo de 2π, tenemos que $e^{-2\pi i k \cdot tr} = 1$, resultando la expresión anterior también igual a 1. Así, si nos quedamos solo con los valores de y múltiplos de r, tenemos:

$$|\psi_4\rangle = \frac{1}{\sqrt{p}} \sum_{t=0}^{p-1} e^{-2\pi i \frac{x_0}{M} \frac{t \cdot M}{p}} |t \cdot \frac{M}{p}\rangle = \frac{1}{\sqrt{p}} \sum_{t=0}^{p-1} e^{-2\pi i x_0 t/p} |t \cdot \frac{M}{p}\rangle$$

La probabilidad de medir cada múltiplo de r es, por tanto, $|\frac{1}{\sqrt{p}}e^{-2\pi i x_0 t/p}|^2$, que es igual a $\frac{1}{p}$. Como tenemos p múltiplos de r, la probabilidad de medir uno de ellos es de 1, por lo que podemos asumir que las amplitudes del resto de $|y\rangle$ se contrarrestan entre sí. Así mismo, la fórmula anterior representa el estado cuántico del circuito de forma completa.

6. Finalmente, midiendo sobre el estado anterior, obtenemos un valor de $t \cdot \frac{M}{p}$ concreto.

Ejecutando varias veces el circuito y obteniendo los suficientes $t \cdot \frac{M}{p}$, podemos calcular p como el máximo común divisor de dichos factores y el módulo (M).

Periodos arbitrarios

Claramente, si solo fuéramos capaces de aplicar el algoritmo anterior con periodos potencias de 2, este no sería capaz de factorizar cualquier número. Sin embargo, ahora estudiaremos qué sucede en el circuito anterior cuando el periodo puede ser cualquier número $1 < p < 2^n$. En este caso, emplearemos $\epsilon = \lceil log_2(n^2) \rceil = \lceil 2 \cdot log_2(n) \rceil$, es decir, el doble de qubits de entrada que en el caso anterior. Paralelamente, consideraremos el módulo $M = 2^\epsilon$.

1. Retomamos el algoritmo anterior por el estado $|\psi_2\rangle$, donde aplicábamos la función. Teníamos entonces:

$$|\psi_2\rangle = \frac{1}{\sqrt{M}} \sum_{x=0}^{M-1} |x\rangle\,|f(x)\rangle$$

En este estado, podemos reagrupar los diferentes x según su valor de $f(x)$ de la siguiente manera:

$$|\psi_2\rangle = \frac{1}{\sqrt{M}} \sum_{j=0}^{r-1} \left(\sum_{x:f(x)=j} |x\rangle \right) |j\rangle$$

En el caso anterior, como p era potencia de 2, teníamos que el número de x que tenían $f(x) = j$ era el mismo para todos los j. Sin embargo, cuando p no tiene esa limitación, esto deja de cumplirse. Concretamente, tendremos $M \ (mod\, p)$ valores de j para los que corresponderán $r+1$ elementos, mientras que el resto de valores tendrán r elementos.

Ahora, calculamos r como $r = \lfloor \frac{M}{p} \rfloor$. Para denotar el cardinal del conjunto de valores x tales que $f(x) = j$ usaremos la notación r_j.

2. Cuando midamos los últimos s qubits, mediremos cierto j concreto y nos encontraremos con el siguiente estado:

$$|\psi_3\rangle = \sqrt{\frac{1}{r_j}} \sum_{k=0}^{r_j-1} |x_0 + k \cdot p\rangle$$

3. Aplicando la $QFT^\dagger$ al estado anterior:

$$|\psi_4\rangle = \frac{1}{\sqrt{M}} \sum_{y=0}^{M-1} \left(\frac{1}{\sqrt{r_j}} \sum_{k=0}^{r_j-1} e^{-\frac{2\pi i y \cdot kp}{M}} \right) \cdot e^{-\frac{2\pi i y \cdot x_0}{M}} |y\rangle$$

   Esta vez, como p no tiene por qué dividir a M, las probabilidades de cada $|y\rangle$ de aparecer no se anulan.

4. Midiendo el estado anterior, podremos obtener cualquier $|y\rangle$ con la siguiente probabilidad:

$$P(y) = \frac{1}{\sqrt{Mr_j}} \left| \sum_{k=0}^{r_j-1} e^{-\frac{2\pi i y k p}{M}} \right|^2$$

   La anterior función de probabilidad tiene sus máximos alrededor, precisamente, de los valores $t\frac{M}{p}$, con $t \in [0, r_j - 1]$. Esto se debe a que, cuando y adquiere dicho valor, los sucesivos números complejos tienen como argumento un múltiplo de 2π, lo cual maximiza el valor del módulo, que será de 1 en dicho caso. Así:

$$P(t\frac{M}{p}) = \left| \frac{1}{\sqrt{Mr_j}} \sum_{k=0}^{r_j-1} e^{-\frac{2\pi i t \frac{M}{p} kp}{M}} \right|^2 = \left| \frac{1}{\sqrt{Mr_j}} \sum_{k=0}^{r_j-1} e^{-2\pi i t k p} \right|^2 = \left| \frac{r_j}{\sqrt{Mr_j}} \right|^2 = \frac{r_j}{M}$$

   Recordando que r_j puede valer $r = \lfloor \frac{M}{p} \rfloor$ o $r + 1$:

$$P(t\frac{M}{p}) \approx \frac{\frac{M}{p}}{M} = \frac{1}{p}$$

   Llegando a la misma probabilidad de cada múltiplo del periodo que en el caso anterior, aunque esta vez de forma aproximada. De la misma manera, considerando que hay r_j valores en $[0, M-1]$ tales que sean múltiplos de $t\frac{M}{p}$, la probabilidad de medir uno de ellos es cercana a 1. Cuanto mayor es el valor de e, esta probabilidad tiende a acercarse asintóticamente a 1.

De esta manera, la salida que nos reporta el circuito cuántico en el caso general es, con alta probabilidad, un número entero muy cercano a $t\frac{M}{p}$. Para encontrar p, se suele dividir el número obtenido $l_t \approx t\frac{M}{p}$ entre M, resultando en la fracción $\frac{l_t}{M} \approx \frac{t}{p}$. De esta aproximación, podemos obtener p empleando el *algoritmo de las fracciones continuas*, que ya vimos en el apartado 5.2.1 para realizar el ataque de Wiener.

6.1.2. Implementación de la exponenciación modular en un circuito cuántico

Una vez visto cómo podemos conseguir el periodo de cualquier función cuántica periódica, nos toca pasar a la implementación del circuito cuántico que defina dicha función. Recordemos que, en el algoritmo de Shor, la función con la que trabajamos es la exponenciación modular, que definimos como:

$$f(x) = a^x \ (mod\ n)$$

Para llegar a dicha función, iremos definiendo diferentes subcircuitos cuánticos con los que implementaremos las operaciones aritméticas clásica necesarias, basándonos en el recurso [VBE96].

Cabe destacar que esta es solo una solución, basada en buena parte en cómo funcionan los circuitos clásicos. Sin embargo, existe otra manera de operar con qubits empleando la Transformada Cuántica de Fourier [Bea03]. Debido a la extensión del trabajo, nos limitaremos a citar esta última.

Suma

Para la implementación de la suma, podemos proceder de forma similar a los circuitos lógicos clásicos. No obstante, todas las funciones que implementemos deben ser reversibles para que sean fácilmente ejecutables en un ordenador cuántico, lo cual complica un poco la implementación.

Antes de pasar directamente a implementar la suma de N qubits, comenzaremos definiendo los dos subcircuitos cuánticos simples en los que podemos descomponerla: el primero realizará el acarreo de la suma de tres qubits ($CARRY$), y el segundo la suma de dos qubits (SUM). Estos circuitos se implementan fácilmente con puertas CNOT y Toffoli, como se muestra en la figura 6.5.

A partir de estas, la idea del circuito de la suma S de dos registros $|A\rangle = |a_n..a_1a_0\rangle$ y $|B\rangle = |b_n..b_1b_0\rangle$ de N qubits busca dar como salida A y $A + B$ (este último de $N + 1$ qubits). Para realizar las operaciones,

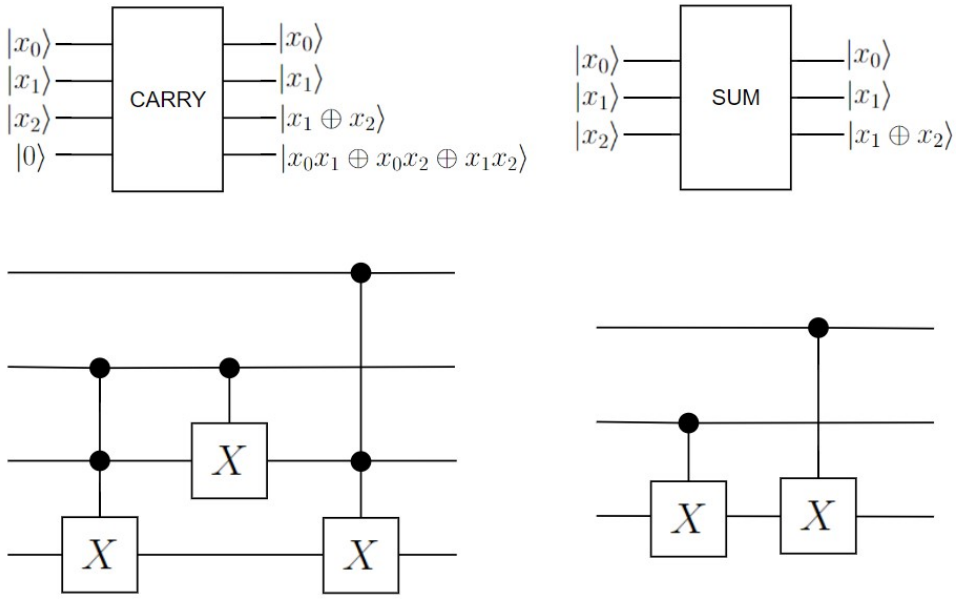

Figura 6.5: Subcircuitos cuánticos *CARRY* (izquierda) y *SUM* (derecha).

necesitaremos $N-1$ qubits auxiliares $c_{1..n}$[1] donde almacenar los acarreos, que almacenaremos en el registro C. Nótese que, aunque no devolvemos como salida B, la función sigue siendo reversible, pues se puede calcular la entrada original realizando la resta $(A+B)-A$.

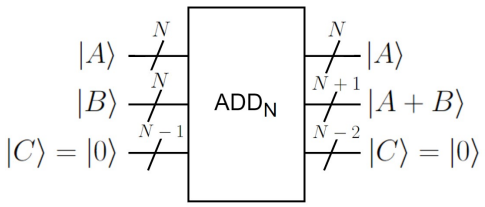

Figura 6.6: Entradas y salidas del circuito ADD_N.

A continuación, vamos a describir los pasos a seguir para implementar un sumador de N qubits:

1. Calculamos el qubit $N+1$ de la suma, es decir, el más significativo. Este lo conseguiremos calculando los sucesivos acarreos aplicando circuitos *CARRY* desde los qubit menos significativos hacia los más significativos. A la par que se calculan los acarreos, también vamos

[1]Como el acarreo c_0 siempre es 0, puede reusarse algún valor de c_i en sustitución a este.

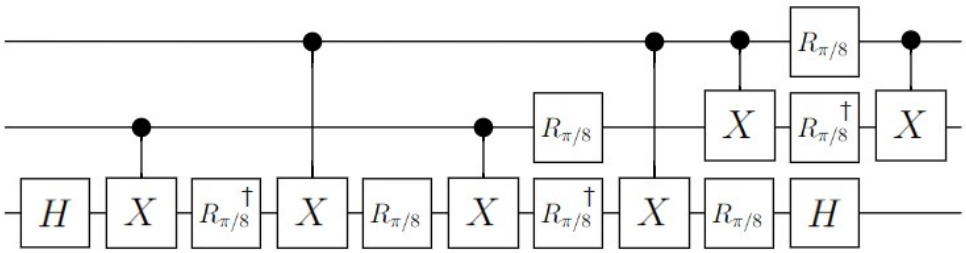

Figura 6.7: Implementación de la puerta de Toffoli en Qiskit, a partir de puertas de 1 y 2 qubits.

obteniendo en el registro B los sucesivos $a_i \oplus b_i$, los cuales tendremos que resetear posteriormente.

2. Una vez obtenemos s_{n+1} y los diferentes acarreos c_i, calculamos el resto de qubits de la suma a la par que reseteamos el valor de los acarreos. Para ello:

   - Aplicamos sobre el qubit $b_n = a_n \oplus b_n$ un CNOT controlado por a_n, obteniendo de vuelta b_n.
   - Colocamos el circuito de suma sobre cada c_i, a_i y b_i y calculamos $s_i = a_i \oplus b_i \oplus c_i$ sobre qubit b_i.
   - Después de cada circuito de suma i (excepto para $i = 0$), usamos el circuito inverso de $CARRY$ sobre c_{i-1}, a_{i-1}, b_{i-1} y c_i, devolviendo a estos dos últimos qubits sus valores originales: $|b_{i-1}\rangle$ y $|0\rangle$, respectivamente. Este circuito inverso no es más que el mismo circuito $CARRY$ pero con sus puertas aplicadas en orden inverso.

Un sumador de N qubits tiene, por ende, $2N - 1$ subcircuitos $CARRY$ o $CARRY^{\dagger}$, N subcircuitos SUM y una puerta X. Para realizar el conteo de puertas cuánticas usadas, tendremos en cuenta que las puertas de 1 y 2 qubits son elementales, puesto que estas son las que implementan los ordenadores cuánticos reales de forma nativa. Para crear cada puerta de Toffoli, necesitaremos 15 puertas cuánticas elementales (véase la implementación dada por Qiskit [Qis24], representada en la figura 6.7). En consecuencia, cada subcircuito $CARRY$ consta de 31 puertas y cada SUM cuenta con 2, llegamos a que necesitamos $64N - 31$ puertas para implementar el circuito ADD_N.

En la figura 6.8 podemos ver un ejemplo de un sumador de 2 qubits implementado con las indicaciones anteriores. Cabe destacar que, al implementar la suma de forma reversible, si aplicamos el circuito cuántico inverso

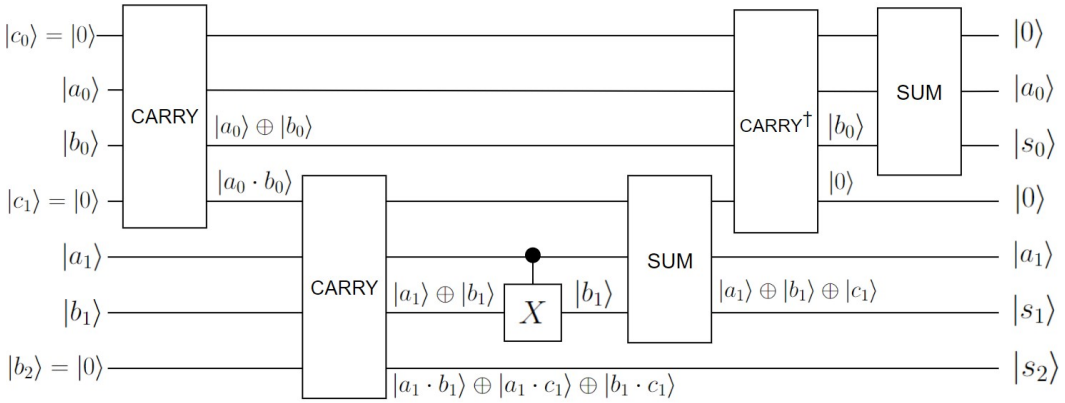

Figura 6.8: Implementación del circuito sumador de números de dos qubits ADD_2.

obtenemos la operación inversa a la suma: la resta. Usaremos esta propiedad para implementar los siguientes circuitos.

Suma modular

La siguiente operación que implementaremos es la suma modular. En la figura 6.9 podemos ver la definición de las entradas y salidas de este circuito.

En primer lugar, tenemos las entradas a sumar A y B, y el módulo M sobre el que realizar el cálculo. Después, tenemos el qubit auxiliar O y el registro auxiliar C, necesario para aplicar el circuito de la suma internamente. En total, nuestro circuito trabaja entonces con $4N$ qubits.

En cuanto a salidas, estas coinciden con las entradas a excepción del registro B, que adquiere el valor de la suma módulo M de A y B. De nuevo, el circuito resulta reversible, pues podemos reconstruir B calculando $(A + B) - A \ (mod\ M)$.

Para realizar el cálculo de la misma de forma eficiente, impondremos una condición sobre las entradas A y B: la suma de ambas debe ser menor que $2 \cdot M$, siendo M el módulo con el que hacemos la suma modular.

Dada dicha condición, podemos calcular $A+B \ (mod\ M)$ sumando $A+B$ y restando M si $A+B > M$. Como $A+B < 2M$, sabemos que $A+B-M < M$, por lo que nunca será necesario realizar más de una resta de M a la suma $A + B$. Como en el algoritmo de Shor realizaremos los cálculos $mod\ M$, siempre tendremos que $A < M$ y $B < M$, y por tanto $A + B < 2M$, así que no nos supone una limitación.

Construiremos el circuito conociendo a priori el valor de la entrada M.

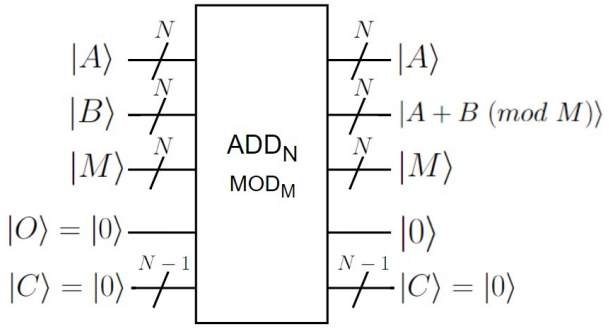

Figura 6.9: Entradas y salidas de la suma modular de N qubits.

También es posible crear el circuito con M como parámetro de entrada, pero ello conllevaría la utilización de más puertas cuánticas y una mayor complejidad, las cuales no deseamos.

El circuito que implementa la suma modular se describe en la figura 6.10. Vamos a analizar las operaciones que realiza el mismo:

1. En primer lugar, se calcula la suma de $A + B$, y acto seguido se les resta el módulo M, resultando en el estado $|S\rangle = |A + B - M\rangle$.

2. Si nos fijamos en el qubit más significativo de $|A + B - M\rangle = S_{N+1}$, observaremos que representa si la resta ha causado desbordamiento o no.

   - Si $A + B < M$, entonces la resta producirá desbordamiento, y almacenaremos en el qubit $|O\rangle$ el valor $|0\rangle$.
   - Si $2M > A + B \geq M$, entonces la resta no producirá desbordamiento, y almacenaremos en el qubit $|O\rangle$ el valor $|1\rangle$.

3. Dependiendo del valor de $|O\rangle$, realizaremos una segunda suma donde:

   - Si $|O\rangle = |0\rangle$, entonces se suma M a $|S\rangle$, dándonos $|A + B\rangle$.
   - Si $|O\rangle = |1\rangle$, entonces se activan una serie de puertas $CNOT$ que borran M de la entrada, sustituyéndolo por $|0\rangle$. Como sabemos M a priori, esto equivale a aplicar una puerta X sobre aquellos qubits con valor $|1\rangle$ que codifican el $|M\rangle$. Así, se suma 0 a $|S\rangle$, dándonos $|A + B - M\rangle$.

   Como resultado, habremos obtenido en ambos casos $A + B\ (mod\ M)$.

4. Para resetear el qubit $|O\rangle$, necesitamos volver a calcular el desbordamiento. Para ello, restamos A al resultado.

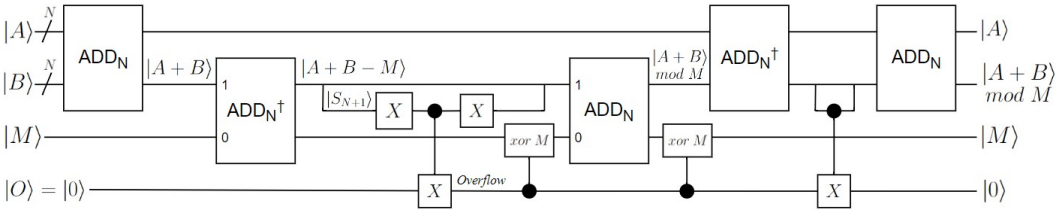

Figura 6.10: Implementación del circuito sumador de dos números de N qubits módulo M ($ADD_N\ MOD_M$).

- Si obtenemos un resultado negativo y el qubit $|S_{N+1}\rangle = |1\rangle$, eso significa que hubo desbordamiento antes también, y que $|O\rangle = 1$.
- Si obtenemos un resultado positivo, con $|S_{N+1}\rangle = |0\rangle$, eso significa que no hubo desbordamiento antes y $|O\rangle = 0$.

Por tanto, aplicando un $CNOT$ con control en $|S_{N+1}\rangle$ sobre $|O\rangle$ conseguimos limpiar dicho qubit.

5. Finalmente, volvemos a sumar A y conseguimos $A + B\ (mod\ M)$.

Veamos cuántas puertas elementales emplea nuestro circuito. Usamos:

- 5 circuitos ADD_N, cada uno con $64N - 31$ puertas.
- 2 subcircuitos $xor\ M$, que contienen tantas puertas X como bits a 1 tiene la representación de M en binario. En el peor caso, usaremos $2N$ puertas X para implementar ambos circuitos.
- 4 puertas X independientes.

Realizando la suma, nuestro circuito contiene $322N - 151$ puertas cuánticas. También cabe contemplar la necesidad de introducir M como entrada al circuito, lo cual puede suponer usar hasta M puertas más.

Multiplicación modular controlada

Tras implementar la suma modular, el siguiente paso será la multiplicación modular. Concretamente, definiremos un circuito cuántico que, dada una entrada $|X\rangle$, calcule $|a \cdot X\ (mod\ M)\rangle$ para un $|a\rangle$ y $|M\rangle$ predefinidos.

Este circuito tiene como entradas el registro de N qubits $|X\rangle$ a multiplicar por a, otro registro $|Y\rangle$ donde escribir la salida, inicialmente a 0, y un qubit de control $|c\rangle$, el cual controlará la ejecución de la multiplicación. Este último qubit nos interesa especialmente de cara a montar el circuito de

la exponenciación modular rápida. Además, usaremos $2N$ qubits auxiliares que nos permitirán ejecutar los subcircuitos usados (N como entrada A a los sumadores, N para almacenar $|M\rangle$, 1 para $|O\rangle$ y otros $N-1$ para los acarreos de la suma).

Para la implementación de la misma, descomponemos la multiplicación modular en N sumas modulares de $2^i a \cdot x_i \ (mod\ M)$, siendo x_i el valor del qubit i del registro X. Como conocemos el valor de a, podemos calcular los diferentes $2^i a$ para $i = 0..N$ y sumarlos o no al resultado en función de si el qubit de control c está activo y x_i es $|1\rangle$ o $|0\rangle$, respectivamente. Para realizar esto, tendremos N circuitos sumadores modulares en serie. Justo antes de cada uno se colocarán puertas de *Toffoli* para situar el sumando necesario en cada paso (bien $2^i a \ (mod\ M)$, bien 0), y después de la suma se volverán a aplicar las mismas puertas para borrar el sumando usado.

Una característica que nos será de utilidad en el siguiente paso es hacer que si el qubit de control está desactivado, entonces la salida del circuito sea igual a la entrada. Para ello, "copiamos" la entrada X en la salida Y usando puertas de *Toffoli* con control en la negación de $|c\rangle$ (alcanzable aplicando una puerta X, acordándonos de deshacer luego el cambio) y cada uno de los qubits que forman $|X\rangle$.

En la figura 6.12 podemos ver el circuito de multiplicación modular que hemos descrito. Dado que:

- Cada una de las N sumas modulares de N qubits requieren como mucho de $322N - 151$ puertas elementales.
- Los $2N+1$ movimientos de qubits entre registros requieren como máximo N puertas de Toffoli (de 15 puertas básicas cada una).
- Se necesitan dos puertas X para negar el qubit $|c\rangle$ antes y después del último movimiento de registros.

el número de puertas que emplea este nuevo circuito es de $352N^2 - 140N + 2$ en el peor caso ($O(N^2)$).

Exponenciación modular rápida

Finalmente, a partir de la multiplicación modular controlada, seremos capaces de implementar la exponenciación modular rápida. De nuevo, este circuito se construirá para una base a concreta, y a partir de un número X calculará $a^X \ (mod\ M)$.

Por tanto, nuestro circuito tendrá como entradas el exponente $|X\rangle$, de N qubits; otros N qubits dedicados a la salida, que inicialmente estarán en

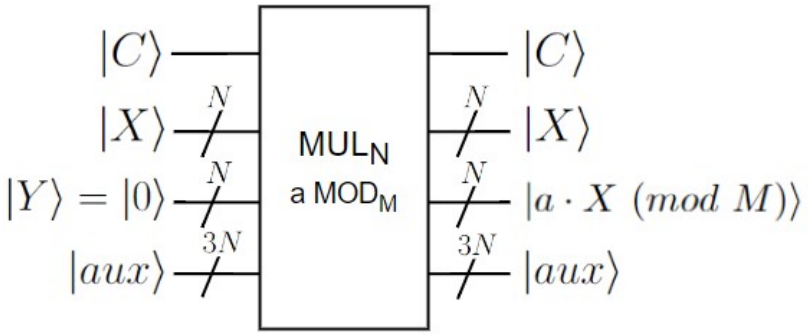

Figura 6.11: Entradas y salidas de la multiplicación modular por a de N qubits.

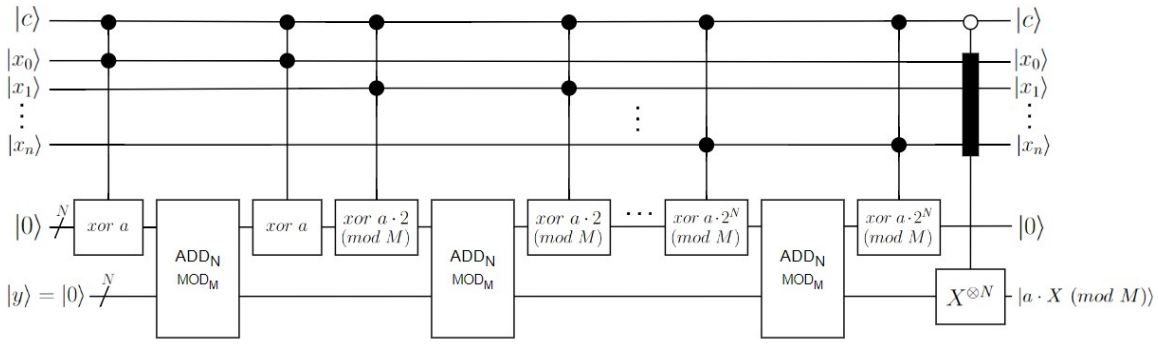

Figura 6.12: Implementación del circuito de multiplicación controlada por $|c\rangle$ de un número $|X\rangle$ por una constante a.

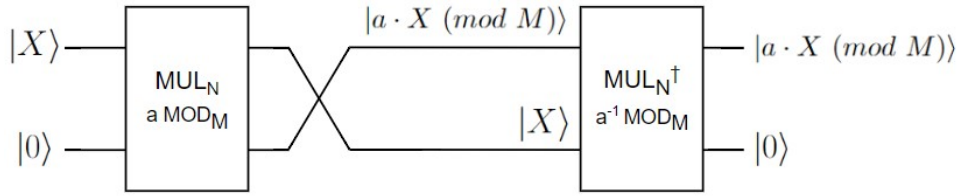

Figura 6.13: Limpieza de registros en multiplicaciones modulares. Como $a^{-1}a \cdot X \ (mod\ M) \equiv X \ (mod\ M)$, si introducimos como entrada $|a \cdot X \ (mod\ M)\rangle\,|0\rangle$ al circuito $MUL_N\ a^{-1}\ MOD_M$ (sin invertir), obtendremos el estado $|a \cdot X \ (mod\ M)\rangle\,|X\rangle$. Además, como el circuito de la multiplicación es reversible, si invertimos el circuito y le damos como entradas las salidas obtenidas antes, recuperamos el estado de entrada $|a \cdot X \ (mod\ M)\rangle\,|0\rangle$.

el estado $|0\rangle$; y $4N$ qubits auxiliares que nos permitirán ejecutar la multiplicación modular.

Como salidas, esperamos obtener las mismas entradas que introducimos en el circuito excepto en el registro de salida, donde obtendremos el estado $|a^X \ (mod\ M)\rangle$.

Para realizar el cálculo de esta salida, procederemos de manera similar al algoritmo de exponenciación rápida: comenzaremos con el valor 1, e iremos multiplicando dicho valor por $a^{2^i} \ (mod\ M)$ si $|x_i\rangle = |1\rangle$.

En términos de circuitos cuánticos, podemos construir el multiplicador modular de cada $a^{2^i} \ (mod\ M)$ y colocarlos en cascada, de forma que la salida de uno sea la entrada del siguiente. Como conocemos a a priori, podemos calcular los diferentes $a^{2^i} \ (mod\ M)$ de forma clásica y crear los circuitos correspondientes. Sin embargo, cada vez que usamos un multiplicador, cambiamos el número a multiplicar en la siguiente iteración, quedando la entrada A de cada multiplicador i sin uso tras la ejecución del multiplicador, y pendiente de ser devuelta a su valor original.

Para solucionar este problema, también construiremos los circuitos inversos a las multiplicaciones modulares de cada inverso de $a^{2^i} \ (mod\ M)$, y los aplicaremos después de cada multiplicación modular tal y como indica la figura 6.13. Así, podemos limpiar los registros y trabajar siempre con los mismos qubits.

Una vez resuelto este problema, lo único que falta por mencionar es que cada par de multiplicaciones-limpiezas de registro van controladas por su correspondiente $|x_i\rangle$. Si está activado, el circuito calculará efectivamente la multiplicación. En caso contrario, la salida de la primera multiplicación es igual a la entrada (pues así lo definimos cuando la implementamos), llegando al circuito inverso la entrada $|X\rangle\,|X\rangle$. Como este circuito también tiene

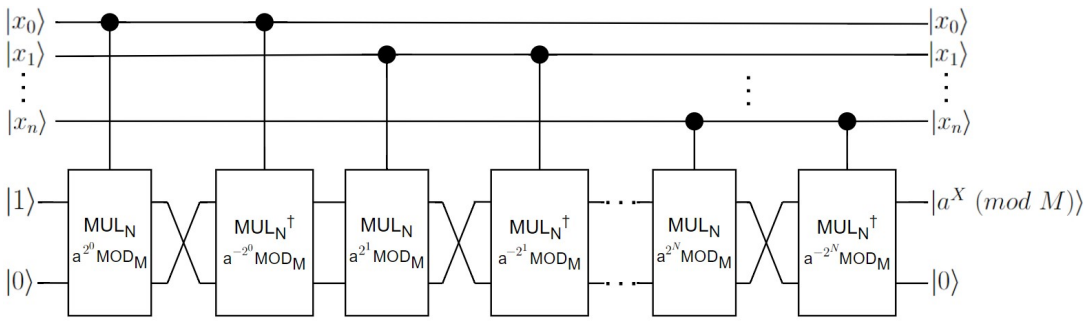

Figura 6.14: Implementación del circuito de exponenciación modular rápida para la base a y módulo M.

el qubit de control deshabilitado, en su versión sin invertir también devolvería $|X\rangle\,|X\rangle$ dada la entrada $|X\rangle\,|0\rangle$. Por tanto, al aplicarlo al revés, se deshace el cambio correctamente, transmitiendo el valor $|X\rangle$ a los siguientes multiplicadores sin alterarlo.

Para montar este circuito, por ende, necesitaremos $2N$ circuitos de multiplicación modular controlada. Atendiendo a que cada uno de ellos tiene como máximo $352N^2 - 140N + 2$ puertas cuánticas elementales, podemos acotar el número de puertas cuánticas necesarias en $702N^3 - 280N^2 + 4$, lo cual es del orden de $O(n^3)$ puertas cuánticas.

En conclusión, nuestro circuito emplea una cantidad polinómica de puertas para calcular la función $f(x) = a^x \ (mod\ N)$, necesaria para aplicar el algoritmo de Shor.

Capítulo 7

Protocolos de intercambio de claves por canales cuánticos

Como hemos visto en el capítulo anterior, la computación cuántica augura el fin de la seguridad de RSA tras casi 50 años desde su concepción. Sin embargo, también surgen de ella nuevas formas de criptografía.

Una de las principales propuestas de la computación cuántica en la criptografía es el intercambio de claves por canal cuántico. Esta consiste en generar, mediante un protocolo, una cadena de bits aleatoria que únicamente conozcan dos interlocutores. Una vez generada esta secuencia, ambos podrán comunicarse de forma perfectamente segura empleando dicha clave como libreta de un solo uso, o empleando un algoritmo de cifrado simétrico seguro con dicha clave.

En este capítulo, examinaremos algunos de los algoritmos más conocidos de intercambio de claves que hacen uso de ordenadores y canales cuánticos, justificando su seguridad ante el potencial espionaje de la comunicación por parte de terceros.

7.1. BB84

En 1984, Bennett y Brassard propusieron un algoritmo de compartición de claves cuántico, el cual fue conocido posteriormente como BB84 [BB14]. Este algoritmo se aprovecha del hecho de que la medición de un qubit altera su estado para asegurarse de que una clave transmitida es secreta y que nadie aparte del emisor y el receptor ha accedido a ella. El algoritmo sigue los siguientes pasos:

1. El emisor, Alice, genera un mensaje m de n qubits.

2. Para cada qubit del mensaje, Alice aplica con un 50 % de probabilidad una puerta H. De esta forma, la mitad de los qubits permanecerán inalterados y la otra mitad pasarán a estar en los estados de la base de Bell ($|0\rangle$ pasa al estado $|+\rangle$, mientras que $|1\rangle$ se convierte en $|-\rangle$).

3. Alice manda al receptor, Bob, los qubits que ha generado a través de un canal cuántico.

4. Cuando el mensaje llega a Bob, este escoge aleatoriamente en qué eje medir cada qubit: si en el eje X o en el eje Z. Cabe destacar que medir en el eje X equivale a aplicar una puerta de Hadamard al qubit antes de medirlo en el eje Z. Una vez elegida la manera en que medir cada qubit, Bob aplica las mediciones, obteniendo el mensaje medido m'.

5. Aquí, Bob publica la forma en que ha medido el mensaje m y Alice comparte a qué qubits ha aplicado puertas H. Aquí se nos presentan dos casos:

   - Si Alice no ha usado una puerta H sobre el qubit i y Bob ha medido dicho qubit en el eje Z, entonces Bob obtendrá el valor correcto para dicho qubit. Lo mismo ocurre si Alice sí emplea la puerta H y Bob mide en el eje X: Bob medirá correctamente el qubit y conocerá su valor original.
   - Si Alice y Bob no coinciden en su elección, entonces la medición que haga Bob tendrá un 50 % de posibilidades de ser correcta.

   En ausencia de errores, se descartan los qubits correspondientes al segundo caso y formamos una clave con los valores transmitidos correctamente.

Supongamos que un tercero, Eve, trata de leer los qubits durante la transmisión del mensaje. Entonces, el mensaje solo se verá inalterado si el espía elije para todos los qubits la base de medición correcta (X o Z). En otro caso, se modificarán los estados de los qubits donde Eve mida en el eje equivocado, cambiándolos de base:

- Si Eve mide un qubit $|0\rangle$ o $|1\rangle$ en el eje X, el estado resultante será $|+\rangle$ o $|-\rangle$, ambos con la misma probabilidad.
- Si Eve mide un qubit $|+\rangle$ o $|-\rangle$ en el eje Z, el qubit colapsará a $|0\rangle$ o $|1\rangle$, cada uno con la misma probabilidad.

Si Alice ha elegido cómo representar cada qubit al azar de manera aleatoria, entonces esperamos que la observación por parte del intermediario altere el 50 % de los qubits de esta manera.

Cuando Bob recibe el mensaje perturbado y realiza sus mediciones, pueden darse varios casos:

- Si Bob elije una medición diferente a la forma en que Alice ha representado el qubit, el resultado esperado es del 50 % de probabilidades para ambos estados elementales, por lo que el espía, independientemente de lo que haya hecho, no será detectado en este caso.
- Si Bob elije la misma medición que Alice, entonces ambos esperan obtener los mismos resultados al medir los qubits. Entonces:
  - Si Eve mide con la misma puerta que ambos, no cambiará el estado del qubit y tampoco será detectado.
  - Sin embargo, si falla, alterará el estado del qubit a uno de la otra base. Entonces, Bob tiene un 50 % de probabilidad de obtener un qubit diferente al que Alice le ha mandado.

La probabilidad de que el atacante sea detectado es, por tanto, de 1/8 por cada qubit que mide, pues se tienen que dar las tres casuísticas anteriores para que ello ocurra (que Alice y Bob midan en la misma base, que Eve mida en otra diferente, y que al medir el qubit este converja al estado opuesto al valor original, todas con una probabilidad individual del 50 %). Para detectar estas diferencias, Alice y Bob publican algunos de los bits del mensaje que han obtenido. Si alguno de estos bits no coinciden, entonces sabrán que alguien ha manipulado los qubits y no usarán la clave compartida. Dado que los bits que conforman la clave provienen de aquellas mediciones donde Alice y Bob miden en la misma base, la probabilidad de que alguno de dichos bits se vea modificado es de 1/4, por lo que si se publican c bits, el atacante será detectado con una probabilidad de $1-\left(\frac{3}{4}\right)^c$. Con 20 bits, la probabilidad de detección se dispara ya al 99,3 %, por lo que es muy difícil que un espía no sea detectado si el mensaje tiene los suficientes bits.

7.2. B92

El protocolo B92 fue propuesto por Charles Bennett en 1992 [Ben92]. La idea que sigue es similar a la de BB84, pero en este protocolo solamente se usan dos estados no ortonormales que forman un ángulo de 90^{o} en la esfera de Bloch. En este trabajo, usaremos $|0\rangle$ y $|+\rangle$. El protocolo se desarrolla como sigue:

1. Alice elige una cadena aleatoria de unos y ceros, y la codifica usando $0 \longrightarrow |0\rangle$ y $1 \longrightarrow |+\rangle$. Hecho esto, envía el resultado a Bob.

2. Bob, al recibir el mensaje de Alice, lee cada qubit del mismo en el eje X o en el eje Z, eligiendo cada eje de manera aleatoria.

   a) Si el qubit recibido es $|0\rangle$ y Bob lo mide en el eje Z, entonces obtendrá el valor 0 con total probabilidad. Sin embargo, si lo mide en el eje X, puede obtener un $|+\rangle$ o un $|-\rangle$, ya que $|0\rangle = \frac{1}{\sqrt{2}}(|+\rangle + |-\rangle)$. Si obtiene un $|-\rangle$, entonces sabrá que Alice le ha mandado el estado $|0\rangle$, pues si le hubiera mandado el estado $|1\rangle$ al aplicar la medición sabe que hubiera obtenido sí o sí un $|+\rangle$.

   b) De forma análoga, si Alice manda un $|+\rangle$ a Bob y lo mide en el eje X, siempre medirá el estado $|+\rangle$, pero si lo hace en Z, existe la posibilidad de que obtenga el estado $|1\rangle$ al medir, lo cual sería imposible si el qubit mandado hubiera sido $|0\rangle$. Por ello, Bob sabrá que el qubit original era $|+\rangle$.

3. Finalmente, Bob comunica a Alice qué qubits son los que conoce con total confianza del proceso anterior, y se descartan el resto. Así, ambos obtienen una clave secreta.

4. Para verificar que no haya ningún espía escuchando, se realiza un proceso similar a BB84: se comparten algunos qubits de la clave y si hay alguno diferente se detecta el ataque.

Está claro que si Eve lee alguno de los qubits en el eje Z y obtiene $|0\rangle$, no podrá distinguir si el qubit original era realmente $|0\rangle$ o $|+\rangle$. Lo mismo ocurre si mide en X y obtiene un $|+\rangle$. Entonces, no será capaz de reconstruir la información correctamente para que la mida Bob, originando las diferencias detectables por los interlocutores.

Una estrategia óptima para minimizar la alteración de qubits por parte de Eve sería la siguiente:

1. Si Eve mide en el eje Z un $|0\rangle$, entonces lo más probable es que fuera un $|0\rangle$ originalmente, con una probabilidad de 2/3. Por tanto, no lo modificará al redirigirlos a Bob.

2. Si Eve mide en el eje Z un $|1\rangle$, entonces sabrá que originalmente dicho qubit era $|+\rangle$. Por tanto, aplicará a su salida las puertas X y H para reconstruir el $|+\rangle$ original.

3. Si el espía mide en el eje X y obtiene un $|+\rangle$, lo más probable es que originalmente dicho qubit fuera $|+\rangle$. Por ello, no lo modifica.

4. Si por el contrario se obtiene $|-\rangle$, Eve sabrá que originalmente el qubit era $|0\rangle$, y aplicará una puerta H seguida de una X para obtenerlo.

Siguiendo este plan, Eve solo fallará reconstruyendo los qubits cuando mida como $|0\rangle$ el estado de superposición $|+\rangle$ y cuando mida como $|+\rangle$ el estado $|0\rangle$ en el eje X, lo cual supone un 25 % de los casos. Por tanto, podemos esperar que la reconstrucción salga mal una vez por cada cuatro qubits del mensaje.

En consecuencia, cuando dicho mensaje alterado llegue a Bob, la probabilidad de que Bob obtenga un 1 al medir cada qubit y que este forme parte de la clave es también de 1/4. Así, lo esperado es que Bob detecte al espía con una probabilidad de 1/16 por cada qubit enviado en el protocolo, y que 1/4 de los bits que formen la clave sean erróneos.

7.3. E91

En 1991, Artur Ekert realiza otra propuesta de protocolo cuántico de intercambio de claves [Eke91]. Este se basa en el uso de pares de qubits entrelazados y de un resultado conocido como el *teorema de Bell*. Para explicar estos conceptos, las referencias [Ber19] y [Exc22] han sido de gran utilidad.

El teorema de Bell

A principios del siglo XX, hubo un fuerte debate entre físicos sobre cómo se comportan las partículas en el mundo cuántico.

Por un lado, la mayor parte de ellos, entre los cuales podemos destacar a Einstein y Schrödinger, defendían la existencia de un sistema subyacente que explicara los comportamientos y resultados contraintuitivos y probabilísticos de la física cuántica. De aquí surge la famosa cita de Einstein: "*Dios no juega a los dados con el universo*". En este bando, argumentaban que las mediciones resultantes al medir dos partículas cuánticas entrelazadas venían ya determinadas desde el momento del entrelazamiento. Cuando se realizaba un entrelazamiento, las partículas interactuaban localmente de una manera desconocida, guardando cierta información entre ellas sobre el resultado que adquirirán al medirse en cada eje posible. Por ello, la teoría de estos se denominó *teoría de variables ocultas*, pues estas variables no se correspondían con observables físicos, lo cual las lleva a estar siempre ocultas.

Por el contrario, físicos como Bohr pensaban que el entrelazamiento sí es un proceso a distancia: el estado de ambos qubits colapsará únicamente una vez midamos uno de los qubits entrelazados, y no antes.

Para resolver este debate, se propuso el siguiente experimento: imaginemos que Alice y Bob tienen cada uno una parte de un par EPR $\psi = \frac{(|00\rangle+|11\rangle)}{\sqrt{2}}$. Entonces, Alice medirá su qubit en uno de tres posibles ejes ($0^{\text{o}} \equiv Z$, 120^{o} o 240^{o}), elegido aleatoriamente con probabilidad uniforme. Di-

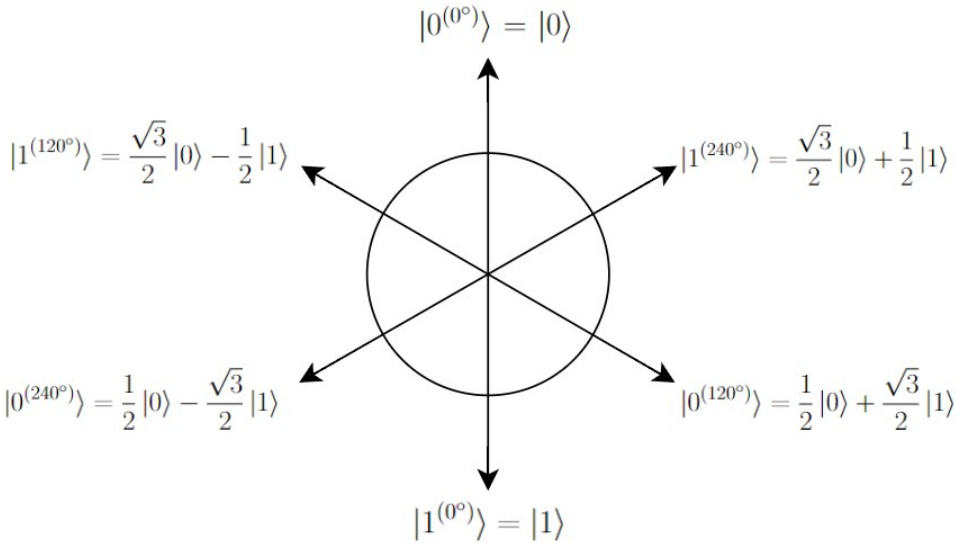

Figura 7.1: Estados ortonormales de los ejes 0º, 120º y 240º considerados en el experimento de Bell. Todos los ejes se encuentran en el plano XZ de la esfera de Bloch.

chos ejes vienen definidos por sus estados ortonormales en la figura 7.1. Una vez realizada la medición, Alice guarda el resultado y Bob mide su qubit en alguno de los tres ejes anteriores, apuntando también el resultado. Este proceso se repite hasta obtener una ristra de ceros y unos de tamaño significativo. Entonces, se comparan ambas cadenas y se anota cuántas veces han coincidido las mediciones y cuántas no.

Dependiendo del modelo que consideremos, Bell se dio cuenta de que se obtienen diferentes proporciones de aciertos.

- Si escogemos la física cuántica, entonces tenemos inicialmente el estado:
$$\psi = \frac{(|00\rangle + |11\rangle)}{\sqrt{2}} = \frac{(|aa\rangle + |bb\rangle)}{\sqrt{2}}$$
siendo a, b cualquier pareja de estados ortonormales (es fácil hacer las cuentas y ver que, si se cambia de base el qubit, siempre obtendremos dicho resultado).

  Una vez Alice mide dicho estado en un eje aleatorio, el estado colapsará a uno de los estados elementales a o b de dicho eje. Entonces, Bob elegirá otro eje al azar y medirá el qubit. Si Bob elige la misma base que Alice, entonces los resultado coincidirán. En otro caso, debemos representar el estado del qubit colapsado por Alice, $|a\rangle$, en la base ele-

gida por Bob, ($|a'\rangle\,|b'\rangle$), para calcular la probabilidad de cada estado. Como los ejes están distanciados entre sí 120^{o}, obtenemos que:

$$|\psi\rangle = |a\rangle = \frac{1}{2}\,|a'\rangle \pm \frac{\sqrt{3}}{2}\,|b'\rangle$$

Resultando en que, si los ejes de medida no coinciden, las medidas de Alice y Bob coincidirán el 25 % de las veces. Juntándolo todo, como la probabilidad de elegir cada eje de medida es de 1/3:

$$P(|a\rangle = |a'\rangle) = \frac{1}{3} + \frac{2}{3}\cdot\frac{1}{4} = \frac{1}{2}$$

- Si seguimos la propuesta clásica, entonces tendríamos que los valores al medir en cada eje de los qubits entrelazados están predefinidos cuando se realiza el entrelazamiento, y por tanto las medidas en un qubit no afectan al resultado en el otro. Entonces, si Alice y Bob eligen el mismo eje para medir, obtendrán el mismo resultado, y si no, las mediciones serán independientes, coincidiendo el 50 % de las veces. Independientemente de la configuración que tenga el par entrelazado, se da que por lo menos la proporción de veces que coinciden las medidas de Alice y Bob es de $P(|a\rangle = |a'\rangle) = \frac{5}{8}$.

Finalmente, sobre los años 60, se comprobó experimentalmente que

$$P(|a\rangle = |a'\rangle) = \frac{1}{2}$$

a través de procedimientos cada vez más sofisticados, los cuales fueron progresivamente descartando las teorías clásicas y apoyando a la teoría cuántica. De esta manera, Bell enunció su teorema como sigue:

Teorema 7.1. ***(de Bell).*** *Ninguna teoría física de variables ocultas locales puede reproducir todas las predicciones de la mecánica cuántica.*

El protocolo E91

Visto el teorema de Bell, podemos directamente explicar y entender los pasos que sigue el protocolo E91:

1. Alice y Bob reciben ambos una parte de varios pares EPR entrelazados, en el estado $\psi = \frac{|00\rangle+|11\rangle}{\sqrt{2}}$.
2. Después, ambos miden sus diferentes pares EPR y anotan tanto el resultado como el eje que han utilizado (que, igual que antes, puede ser el eje 0^{o}, 120^{o} o 240^{o}, y se elige aleatoriamente para cada qubit).

3. Una vez se han realizado las medidas de los qubits, Alice y Bob publican las bases que han usado, conservando los resultados privados. Por probabilidad, sus bases coincidirán un tercio de las veces, y dichos qubits serán iguales para ambos: estos qubits formarán la clave compartida si todo va bien.

4. Finalmente, Alice y Bob publican las medidas obtenidas cuando han elegido bases diferentes. Así, calculan el porcentaje de veces que las medidas han coincidido. Si las operaciones que han realizado han sido sobre pares EPR puros, entonces deberían coincidir en 1/4 de las medidas. Sin embargo, si coinciden en 3/8, habrán detectado la presencia de un espía que ha medido todos los qubits.

Si hay un espía, entonces este deberá realizar mediciones antes de que los qubits lleguen a Alice y Bob. Cuando se hace esto, los qubits que llegan a ambos habrán convergido a cierto estado y perderán sus cualidades cuánticas, alterando la probabilidad de que las mediciones de Alice y Bob concuerden.

Si Alice o Bob eligen el mismo eje de medida que Eve, entonces las probabilidades de que ambos coincidan se mantienen, pues es equivalente a que alguno de los dos haya medido primero el par entrelazado.

La clave está en que si ambos eligen bases diferentes al atacante, entonces los dos tendrán el 25 % de posibilidades de coincidir con la medida de Eve. Por tanto, la probabilidad de que ambos coincidan en este caso es:

$$\frac{3}{4} \cdot \frac{3}{4} + \frac{1}{4} \cdot \frac{1}{4} = \frac{5}{8}$$

De esta manera, conociendo que el anterior caso se dará 1/3 de las veces si todos eligen ejes de forma aleatoria, las probabilidades de que Alice y Bob coincidan son:

$$\frac{2}{3} \cdot \frac{1}{4} + \frac{1}{3} \cdot \frac{5}{8} = \frac{3}{8}$$

por lo que el protocolo detectará correctamente la interferencia de Eve realizando la anterior comprobación.

Parte IV

Implementación y experimentación

Capítulo 8

Implementación de RSA y ataques clásicos

En este nuevo capítulo del trabajo, con el fin de poner a prueba la fortaleza del criptosistema RSA, vamos a tratar sobre la implementación de sus ataques en su faceta clásica.

Para ello, vamos a desarrollar una aplicación de línea de comandos para generar claves RSA. Esta permitirá cifrar y descifrar mensajes. Además, el usuario podrá realizar diferentes ataques a la clave RSA generada para evaluar su seguridad.

8.1. Diseño

En esta sección, vamos a realizar una breve descripción sobre el proceso de diseño del software, abordando temas como el análisis de requisitos, los casos de uso y el flujo del programa.

8.1.1. Análisis de requisitos

A continuación, presentaremos los requisitos que debe satisfacer el software a desarrollar. Comenzaremos con los requisitos funcionales:

1. El sistema debe ser capaz de generar claves RSA de diferentes tipos para estudiar la seguridad de las mismas:
   a) Claves generadas con primos aleatorios.
   b) Claves generadas con primos robustos
   c) Claves generadas con exponente de descifrado vulnerable.

2. El sistema debe permitir el cifrado y descifrado de mensajes de tamaño menor que el módulo de cifra de la clave (n).

3. El sistema debe permitir ejecutar diferentes ataques sobre la clave RSA generada. Los ataques a implementar serán:

   a) Ataque de Fermat.

   b) Ataque de Kraitchik.

   c) Ataque ρ de Pollard.

   d) Ataque $p - 1$ de Pollard.

   e) Ataque de factorización con criba cuadrática.

   f) Ataque de Wiener.

4. El sistema debe implementar un menú por línea de comandos que permita al usuario del programa acceder a las funcionalidades del mismo.

5. El sistema debe medir los tiempos de ejecución de los ataques.

En cuanto a requisitos no funcionales, definir los siguientes:

1. Rendimiento.

   a) El sistema debe poder manejar claves RSA de gran tamaño (8096 bits) eficientemente, siendo el cifrado y descifrado rápidos.

   b) La velocidad a la que se ejecutan los ataques debe ser razonable, según el número a factorizar y el tipo de ataque.

2. Usabilidad.

   a) El sistema debe proporcionar una interfaz fácil de usar.

   b) El sistema debe mostrar detalles sobre la ejecución de los ataques al usuario, tanto con fin ilustrativo del funcionamiento interno del ataque como para el ajuste de parámetros de entrada, en caso de que los haya.

3. Mantenimiento del software.

   a) El código debe estar bien documentado y modularizado.

Finalmente, como requisitos de almacenamiento, solamente necesitaremos guardar una clave RSA. Si se crea una clave y el sistema ya ha generado una clave, se sobrescribirá la clave nueva sobre la anterior.

8.1.2. Casos de uso

Como resultado de los anteriores requisitos funcionales, podemos distinguir cuatro casos de uso de nuestro programa. Estos son crear una clave RSA, cifrar un mensaje con la clave pública, descifrar un criptograma con la clave privada y atacar la clave. Todos ellos tienen como único actor al usuario del programa.

1. Generación de una clave RSA.
   - Precondiciones: Estar en el menú principal.
   - Flujo normal:
     a) El usuario selecciona la opción de generar una clave RSA.
     b) El sistema muestra un submenú con las diferentes opciones de generación de claves.
     c) El usuario selecciona una de las tres opciones.
     d) El sistema genera la clave, la guarda y la muestra por pantalla, volviendo al menú principal.
   - Flujo alternativo: si en $1c$) se elige volver al menú principal, vuelve al menú principal y termina el caso de uso.
   - Postcondiciones: Se ha generado la clave RSA y puede ser usada por otros casos de uso.
2. Cifrado de un mensaje.
   - Precondiciones: Estar en el menú principal.
   - Flujo normal:
     a) El usuario selecciona la opción de cifrar un mensaje.
     b) El sistema pide al usuario introducir el mensaje a cifrar.
     c) El usuario inserta el mensaje por la entrada estándar.
     d) El sistema cifra el mensaje con la clave RSA guardada, muestra por pantalla el resultado y vuelve al menú principal.
   - Flujo alternativo: si en $2a$) se detecta que no hay una clave generada, informa del error y vuelve al menú principal. Termina el caso de uso.
   - Postcondiciones: Se ha cifrado el mensaje con la clave RSA.
3. Descifrado de un mensaje.
   - Precondiciones: Estar en el menú principal.
   - Flujo normal:
     a) El usuario selecciona la opción de descifrar un criptograma.

b) El sistema pide al usuario introducir el criptograma a descifrar.
c) El usuario inserta el criptograma por la entrada estándar.
d) El sistema descifra el criptograma con la clave RSA guardada, muestra por pantalla el resultado y vuelve al menú principal. Termina el caso de uso.

- Flujo alternativo: si en $3a$) se detecta que no hay una clave generada, informa del error y vuelve al menú principal.
- Postcondiciones: Se ha aplicado el ataque sobre la clave y se han mostrado los resultados por pantalla.

4. Ataque a la clave.

- Precondiciones: Estar en el menú principal.
- Flujo normal:
  a) El usuario selecciona la opción de atacar la clave RSA.
  b) El sistema muestra los diferentes ataques disponibles.
  c) El usuario selecciona uno de los ataques disponibles.
  d) Si el ataque seleccionado tiene parámetros de entrada:
    1) El sistema solicita los parámetros al usuario.
    2) El usuario escribe los parámetros en el programa.
  e) El sistema ejecuta el ataque seleccionado, mostrando su resultado al final de la ejecución y el tiempo de ejecución empleado.
- Flujos alternativos:
  a) Si en $4a$) se detecta que no hay una clave generada, informa del error y vuelve al menú principal. Termina el caso de uso.
  b) Si en $4c$) se elige volver al menú principal, vuelve al menú principal y termina el caso de uso.
- Postcondiciones: Se ha descifrado el criptograma con la clave RSA.

Dada la simplicidad del diagrama de casos de uso resultante, que consistiría en un usuario accediendo a cada caso de uso de manera independiente, hemos decidido no incluirlo, pues consideramos que no aporta valor a esta memoria.

A partir de estos casos de uso, construimos el diagrama de flujo del programa en la figura 8.1. Como el software carece de una alta complejidad, no estimamos oportuna la definición de clases ni proseguir con la concreción de más diagramas de diseño. En este punto, terminamos la fase de diseño y pasamos a la de implementación.

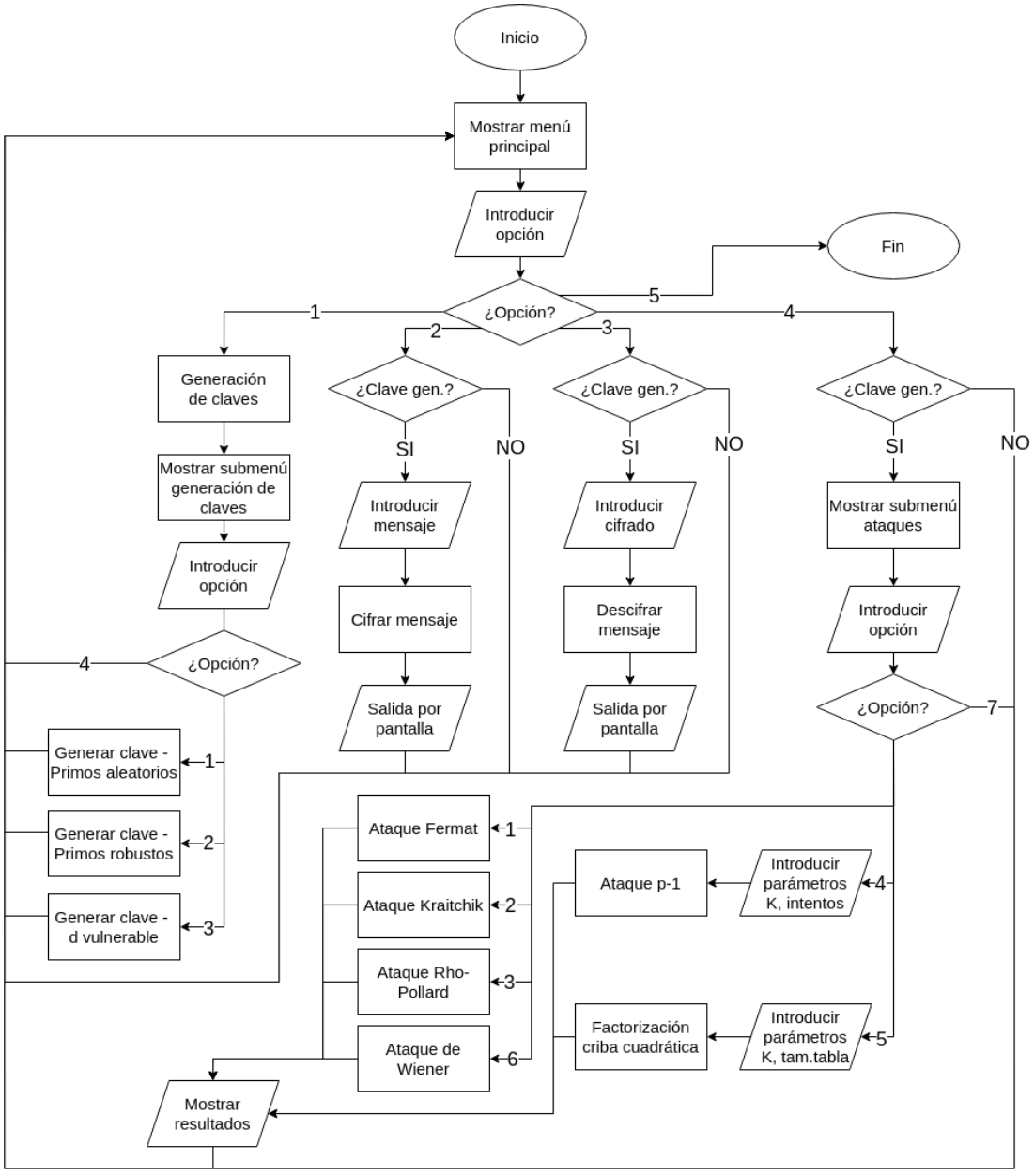

Figura 8.1: Diagrama de flujo del programa.

8.2. Recursos empleados

Para el desarrollo del trabajo, vamos a usar el sistema operativo Linux. El lenguaje de programación usado para esta parte de la práctica es C++. Esta elección se debe a que es un lenguaje que produce programas rápidos, lo cual es un aspecto clave a la hora de ejecutar este tipo de algoritmos; a la par que dispone de las librerías suficientes para realizar un manejo relativamente sencillo de números arbitrariamente grandes.

Concretamente, en este trabajo empleamos las librerías GMP y GMPXX [GNU24]. De ellas, nos interesa especialmente la clase *mpz_class*, que lidia con enteros de un tamaño arbitrario y proporciona ciertas operaciones con ellos, como el cálculo del máximo común divisor, álgebra modular y funciones de teoría de números.

También se emplea GitHub para el control de versiones del proyecto. Gracias a esta herramienta, llevaremos un histórico de la implementación y seremos capaces de revertir cambios en caso de que cometamos errores durante el desarrollo. El repositorio desarrollado (que también contiene la implementación de la parte cuántica, descrita en el capítulo 9) se puede encontrar en el siguiente enlace: https://github.com/jorgelerre/tfg_rsa.

8.3. Estructura del código

La estructura del proyecto consiste en la típica de un proyecto C++: tendremos nuestras carpetas *bin* para los ejecutables, *obj* para los archivos de código objeto, *include* para los archivos de cabecera y *src* para el código fuente. El código fuente se distribuye, con el objetivo de modularizar la implementación, entre cuatro ficheros distintos:

- *rsa.cpp*: Implementa la generación de claves, el cifrado y el descifrado con RSA.
- *rsa_attacks.cpp*: Contiene el código de los ataques al RSA escritos.
- *utils.cpp*: Contiene las funciones auxiliares empleadas por los dos ficheros anteriores. Es la encargada de, por ejemplo, la generación de números primos robustos y de la resolución de ecuaciones cuadráticas.
- main.cpp: Implementa el programa principal, que consta de un menú por línea de comandos que permite acceder a las funciones implementadas.

Los ficheros *rsa.cpp*, *rsa_attacks.cpp* y *utils.cpp* tienen cada uno su respectivo archivo de cabecera en la carpeta *include*. Dichos archivos de cabe-

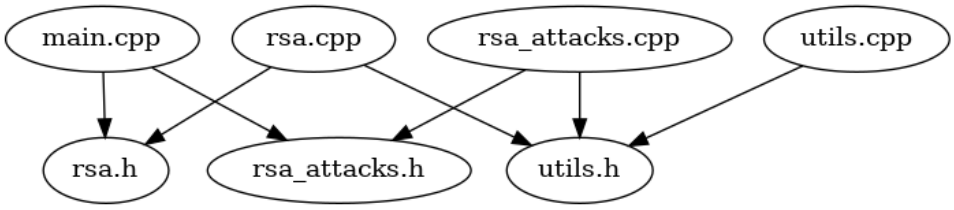

Figura 8.2: Grafo de dependencias del proyecto.

cera contienen una breve descripción de cada una de las funciones implementadas. Allí se explican las acciones que realiza, los parámetros que toma de entrada y las salidas que ofrece.

Además, se incluye en la carpeta raíz del proyecto un *Makefile*, el cual permite compilar automáticamente el código ejecutando el comando *make* en el terminal desde dicho directorio. Este generará, si no existen, las carpetas *obj* y *bin* y creará el ejecutable en *bin/main*.

8.4. Implementación de RSA

Para representar una clave RSA, vamos a utilizar tres objetos de la clase *mpz_class*. Esta clase hace de envoltorio del tipo de dato *mpz_t*, que representa un número entero de tamaño arbitrario. Los tres objetos harán de módulo de cifra n, clave de cifrado e y clave de descifrado d.

La implementación de la generación de claves de RSA se ha realizado, tal y como hemos especificado en el primer caso de uso, atendiendo a dos factores: si queremos generar la clave con primos robustos, o si queremos usar un exponente de descifrado bajo. En consecuencia, vamos a implementar una función, llamada *generaClaveRSA*, encargada de generar la clave correspondiente en función de los dos factores mencionados.

Para crearla, es necesario disponer de un generador de primos, tanto aleatorios como robustos. Por ello, en *utils.cpp* implementamos ambos generadores junto con sus dependencias:

- *strongPseudoprime*, que aplica el test de si un objeto de la clase *mpz_class* es pseudoprimo robusto en otra base.
- *millerRabin*, que aplica el test de Miller-Rabin sobre un objeto de la clase *mpz_class* con cierto número de pruebas k. Para ello, emplea la función *strongPseudoprime* en cada iteración con diferentes bases.
- *generatePrime*, la cual genera primos de cierto número de bits usando

la función *millerRabin*.

- *generateStrongPrime*, que genera un primo robusto con el algoritmo visto en el apartado 4.3.3.

El siguiente paso en la implementación después de programar las anteriores funciones es, ahora sí, programar la generación de claves de RSA. Esta se representa en el algoritmo 1.

Para realizar el cifrado y descifrado, basta con usar internamente la función de exponenciación modular rápida de la librería GMP. En el caso del cifrado, elevaremos un mensaje m al exponente de cifrado e guardado módulo n. Si en vez de eso necesitamos descifrar, la operación es la misma pero cambiando el exponente de cifrado por el de descifrado.

8.5. Implementación de ataques clásicos

Una vez terminada la implementación del propio RSA, vamos a centrarnos en la programación de los ataques contra el mismo.

8.5.1. Ataque de Fermat

Recordemos que el ataque de Fermat busca expresar n como una diferencia de cuadrados $x^2 - y^2 = (x+y)(x-y)$, lo cual factoriza el número.

Para encontrar tales valores de x e y, con el fin de minimizar el número y la complejidad de cálculos en el algoritmo, Fermat plantea el valor $r = (\sqrt{n}+i)^2 - n - j^2$, donde inicialmente $i, j = 0$. Si r es mayor que 0, entonces reduce el valor de r sumando uno a j. Si r es menor que 0, entonces sumamos 1 al valor de i. Sin embargo, si $r = 0 = (\sqrt{n}+i)^2 - n - j^2$, entonces podemos despejar $n = (\lfloor\sqrt{n}\rfloor + i)^2 - j^2$ y obtener la diferencia de cuadrados buscada.

No obstante, calcular potencias en cada iteración es costoso, así que aquí se plantean dos nuevos valores, u y v, los cuales representarán la diferencia actual entre los cuadrados $(\lfloor\sqrt{n}\rfloor + i)^2$ y $(\lfloor\sqrt{n}\rfloor + i + 1)^2$, y j^2 y $(j+1)^2$, respectivamente. La gracia de esto está en que $(k+1)^2 - k^2 = 2k+1$, por lo que en vez de calcular los cuadrados desde cero, podemos realizar la suma de dicha diferencia en su lugar. De este razonamiento sale el algoritmo 2, que es el implementado en el trabajo.

8.5.2. Ataque de Kraitchik

El ataque de Kraitchik, a diferencia del de Fermat, calcula primero $x^2 - n$ y luego comprueba si el resultado es un cuadrado perfecto. Si lo es, entonces

Algorithm 1 Generación de claves RSA

Require: $bits$, $strong\_prime$, low_d
Ensure: Clave generada (n, e, d)

{Usamos primos robustos/aleatorios según el parámetro $strong\_prime$.}
if $strong\_prime$ **then**
  $p \leftarrow$ generateStrongPrime$(bits/2, state)$
  $q \leftarrow$ generateStrongPrime$(bits/2, state)$
else
  $p \leftarrow$ generatePrime$(bits/2, state)$
  $q \leftarrow$ generatePrime$(bits/2, state)$
end if
$n \leftarrow p \cdot q$
$\phi(n) \leftarrow (p-1) \cdot (q-1)$
{Usamos d vulnerable o no en función de low_d.}
if not low_d **then**
  if $\phi(n) > e\_default$ **then**
    $e \leftarrow e\_default$
  else
    $e \leftarrow$ Numero aleatorio menor que $\phi(n)$
    if $e \mod 2 \equiv 0$ **then**
      $e \leftarrow e - 1$ {Hacemos que e sea impar}
    end if
  end if
  $mcd \leftarrow \gcd(e, \phi(n))$
  while $mcd > 1$ **or** $e < 3$ **or** $e > \phi(n)$ **do**
    $e \leftarrow$ Numero aleatorio menor que $\phi(n)$
    if $e \mod 2 \equiv 0$ **then**
      $e \leftarrow e - 1$ {Hacemos que e sea impar}
    end if
    $mcd \leftarrow \gcd(e, \phi(n))$
  end while
  $d \leftarrow e^{-1} \mod \phi(n)$
else
  $d \leftarrow$ Numero aleatorio menor que $\sqrt[4]{n}/3$
  while $d < 2$ **or** $\gcd(d, \phi(n)) > 1$ **do**
    $d \leftarrow$ Numero aleatorio menor que $\sqrt[4]{n}/3$
  end while
  $e \leftarrow d^{-1} \mod \phi(n)$
end if
return (n, e, d)

Algorithm 2 Ataque de Fermat

Require: n (número a factorizar)
Ensure: p (factor encontrado de n)

if MillerRabin(n, 10) **then**
  $p \leftarrow n$ {Si n es primo, termina}
else
  $x \leftarrow \lfloor \sqrt{n} \rfloor$
  if n es un cuadrado perfecto **then**
    $p \leftarrow x$
  else
    $u \leftarrow 2x + 1$
    $v \leftarrow 1$
    $r \leftarrow x^2 - n$
    while $r \neq 0$ **do**
      if $r > 0$ **then**
        $r \leftarrow r - v$
        $v \leftarrow v + 2$
      else
        $r \leftarrow r + u$
        $u \leftarrow u + 2$
      end if
    end while
    $p \leftarrow \frac{u+v-2}{2}$
  end if
end if
return p

habremos encontrado y y podremos factorizar n. El algoritmo 3 muestra el pseudocódigo de la implementación realizada.

Algorithm 3 Ataque de Kraitchik

Require: n (número a factorizar)
Ensure: p (factor encontrado de n)
1: $x \leftarrow 0, sq\_x \leftarrow 0$
2: $p \leftarrow 0$
3: **if** MillerRabin$(n, 10)$ **then**
4: $p \leftarrow n$
5: **else**
6: {Calculamos la raíz cuadrada de n (redondeo hacia arriba)}
7: $x = \lfloor \sqrt{n} \rfloor$
8: **if** n es un cuadrado perfecto **then**
9: $p \leftarrow x$
10: **else**
11: **while** $x < n$ **and** sq_x no es un cuadrado perfecto **do**
12: $x \leftarrow x + 1$
13: $sq\_x \leftarrow x^2 - n$
14: $sq\_x = \sqrt{sq\_x}$
15: **if** sq_x es un cuadrado perfecto **then**
16: $p \leftarrow x + sq\_x$
17: **end if**
18: **end while**
19: **end if**
20: **end if**
21: **return** p

8.5.3. Ataque ρ de Pollard

Respecto a este ataque, su implementación también resulta bastante directa. En el código utilizamos como función pseudoaleatoria $f(x) = x^2 + 1$ mód n, la cual definimos en la función auxiliar *paso_rhoPollard*. Hecho eso, la implementación del método es directa usando el algoritmo 4.

8.5.4. Ataque $p - 1$ de Pollard

Para implementar este ataque, tenemos que introducir una nueva función auxiliar que nos calcule el máximo número tal que sea K-potencia uniforme. Esta función, a la cual llamaremos *maxKPotenciaSuave*, tomará el entero k y calculará dicho número usando el algoritmo visto en el apartado 5.1.2.

Disponiendo de esta, ya estamos en condiciones para programar este

Algorithm 4 Algoritmo ρ de Pollard

Require: n (número a factorizar)
Ensure: p (factor encontrado de n)
1: $mcd \leftarrow 1$
2: **if** MillerRabin$(n, 10)$ **then**
3: $p \leftarrow n$
4: **else**
5: {Tortuga y liebre empiezan en el mismo estado (2)}
6: $t \leftarrow 2$
7: $l \leftarrow 2$
8: **while** $mcd = 1$ **do**
9: {Avanzamos la tortuga un paso}
10: $t \leftarrow$ *paso_rhoPollard(t,n)*
11: {Avanzamos la liebre dos pasos}
12: $l \leftarrow$ *paso_rhoPollard(l,n)*
13: {Calculamos el mcd de la diferencia $t - l$ con n}
14: $mcd \leftarrow \gcd(n, |t - l|)$
15: **end while**
16: $p \leftarrow mcd$
17: **end if**
18: **return** p

ataque usando el esquema mostrado en el algoritmo 5. En este, se toma una base a aleatoria, se calcula a^B y se calcula el máximo común divisor. Este proceso se repite hasta encontrar un factor de n o hasta agotar el número de intentos *att* pasado como argumento.

8.5.5. Ataque de factorización por curvas elípticas

Con el fin de realizar este ataque, debemos dar soporte a las operaciones de suma y multiplicación de una curva elíptica.

Lo primero que haremos será definir el tipo de dato *Punto*, que estará compuesto por dos objetos de la clase *mpz_class*. Como con operaciones modulares no tenemos números negativos, podemos representar el punto O como el punto $(-1, -1)$ y un punto no existente en la curva como $(-2, -2)$.

Dado que trabajamos con grupos cíclicos con tamaño mayor a 3, podemos representar la curva elíptica con dos parámetros a y b. De esta forma, las funciones que realicen operaciones en una curva elíptica recibirán tanto los puntos implicados en la operación como los dos parámetros de la curva.

De esta forma, implementamos la suma de dos puntos de una curva elíptica tal y como se especifica en el apartado 5.1.3, y luego la multipli-

Algorithm 5 Algoritmo $p-1$ de Pollard

Require: n, k, att
Ensure: p (factor encontrado de n)
1: $exito \leftarrow$ **false**
2: $B \leftarrow maxKPotenciaSuave(k)$
3: **for** $i \leftarrow 0$ **to** att **and** $exito =$ **false do**
4: a $\leftarrow$ Numero aleatorio menor que $\phi(n)$
5: $x \leftarrow a^B \mod n$
6: $mcd \leftarrow \gcd(a^B - 1, n)$
7: **if** $mcd < n$ **and** $mcd > 1$ **then**
8: $exito \leftarrow$ **true**
9: **end if**
10: **end for**
11: **return** mcd

cación de un punto por un escalar. Para que resulte eficiente el cálculo de la multiplicación $s \cdot P$, empleamos un esquema similar a la exponenciación binaria: representamos el escalar s en binario y al resultado lo llamaremos s'. Luego, sumamos el punto inicial P consigo mismo, obteniendo $2P$, después $2P$ se suma de nuevo consigo mismo, obteniendo $4P$, y así sucesivamente. De forma paralela, tendremos un punto inicializado con el valor O, al cual le sumamos aquellos 2^iP tales que ${s'}_i = 1$. Cuando hayamos recorrido todo s', tendremos hecho el cálculo $R = \sum_{i=0}^{nbits} 2^iP \cdot {s'}_i = s \cdot P$.

Una característica especial de las implementaciones tanto de la suma como de la multiplicación es que si el punto a calcular no existe, lo cual puede suceder si el módulo de la curva no es primo, entonces devuelve como solución el punto inexistente $(-2, -2)$ y el número *inv* que provocó el error. Este número no tiene inversa en $\mathbb{Z}_n$, por lo que el máximo común divisor del número con n es mayor que 1 y nos devuelve un factor de n.

Usando la aritmética de curvas elípticas y la función *maxKPotenciaSuave*, la implementación es similar al ataque $p-1$ de Pollard.

8.5.6. Ataque de factorización por criba cuadrática

Para terminar con los ataques de factorización, vamos a implementar la criba cuadrática.

En primer lugar, para realizar el proceso de criba, es necesario implementar la raíz cuadrada modular en cuerpos primos. Esto requiere que implementemos el Algoritmo de Tonelli-Shanks [Wik24b], que se encarga justamente de esta operación. Este algoritmo lo implementamos en la función *sqrtMod*.

Después, cuando hayamos calculado los restos k-uniformes y tengamos

Algorithm 6 Factorización de curvas elípticas

Require: n, k, att
Ensure: p (factor encontrado de n)
1: **if** millerRabin(n, 10, $state$) **then**
2: $p \leftarrow n$
3: **else**
4: $L \leftarrow$ maxKPotenciaSuave(k)
5: $exito \leftarrow$ **false**
6: **for** $i \leftarrow 0$ **to** att **and not** $exito$ **do**
7: $a \leftarrow$ Un número aleatorio menor que n
8: $Q_x \leftarrow$ Un número aleatorio menor que n
9: $Q_y \leftarrow$ Un número aleatorio menor que n
10: $b \leftarrow {Q_y}^2 - {Q_x}^3 + a \cdot Q_x$
11: $M \leftarrow$ multiplicacionCurvaEliptica(Q, L, a, b, n, inv)
12: **if** $M \notin E_{a,b}(\mathbb{Z}_n)$ **then**
13: $p \leftarrow \gcd(inv, n)$ {Si el punto no es válido, tenemos un factor}
14: $exito \leftarrow$ **true**
15: **end if**
16: **end for**
17: **if not** $exito$ **then**
18: $p \leftarrow n$
19: **end if**
20: **end if**
21: **return** p

que encontrar una combinación de los mismos que nos dé un resto cuadrático, será necesario un mecanismo de resolución de sistemas de ecuaciones que trabaje en módulo 2. Uno de ellos, que es el que usaremos en este trabajo, es la eliminación gaussiana [Wik24a]. La implementación de esta se divide en dos partes:

- Una función *eliminacionGaussiana* que, dada una matriz, la convierte en una matriz triangular aplicando sobre la matriz original intercambios de filas, multiplicaciones de filas por un escalar diferente de 0 y suma de unas filas con otras.
- Una segunda función *encuentraSoluciones* que, dada la matriz triangular resultante, devuelve todas las soluciones posibles considerando variables libres. Aquí tomamos la precaución de no tomar más de 100 soluciones, puesto que si hay demasiadas variables libres el número de soluciones se dispara y podemos tener problemas de rendimiento gestionando tantos datos.

Con dichas operaciones, implementamos la criba cuadrática siguiendo los algoritmos 7 y 8. En el primero de ellos, calculamos la base de primos, creamos la tabla de cribado y aplicamos el cribado sobre la misma con las dos soluciones que reporta cada primo de la base de primos. Al final, nos quedamos con aquellos restos que sean *k-uniformes*, es decir, cuyo resto en la criba ha terminado siendo 1.

Luego, en el segundo algoritmo, calculamos un resto cuadrático mediante la resolución del sistema de ecuaciones en base 2 de los factores primos de cada resto. Después, para cada solución calculada (sin contar la trivial, que no nos aporta información), calculamos el resto cuadrático obtenido y le aplicamos la raíz cuadrada, mientras que al otro lado de la congruencia calculamos el producto de los x que han sido utilizados para obtener el resto. Finalmente, calculamos el $mcd(x - r, n)$. Si el resultado no es ni 1 ni n, habremos conseguido un factor no trivial de n.

8.5.7. Ataque de Wiener

Por último, programamos el ataque por fracciones continuas de Wiener. Con este fin, creamos dos nuevas funciones auxiliares:

- *cocientesFraccionContinua*, que dado un número racional a/b encuentra su representación en fracción continua. Como dijimos en el apartado 5.2.1, este método es similar al cálculo del Algoritmo de Euclides sobre a y b. La representación en fracción continua de a/b serán los sucesivos cocientes que se obtienen en dicho algoritmo.

Algorithm 7 Factorización Criba Cuadrática (1) - Cribado

Require: n, k, tam_tabla
Ensure: p

$base\_primos \leftarrow [2]$ {Calcular base de primos}
$p\_actual \leftarrow 3$
while $k > p\_actual$ **do**
  $s\_legendre \leftarrow \text{legendre}(n, p\_actual)$
  if $s\_legendre = 1$ **then**
    Añadir p_actual a $base\_primos$
  end if
  $p\_actual \leftarrow$ siguiente primo de p_actual
end while
Calcular $x\_ini \leftarrow \sqrt{n} + 1$
for $i \leftarrow 0$ **to** $tam\_tabla - 1$ **do**
  Añadir $(i + x\_ini)^2 - n$ a $tabla\_criba$ {Creamos la tabla de cribado}
end for
Inicializar $factores$ y $factores\_count$ con ceros
for $i \leftarrow 0$ **to** longitud de $base\_primos - 1$ **do**
  $r1 \leftarrow \text{sqrtMod}(n, base\_primos[i])$ {Calculamos primera solución}
  $sol1 \leftarrow (r1 - x\_ini) \mod base\_primos[i]$
  $it \leftarrow sol1$ {Cribamos con la primera solución}
  while $it <$ longitud de $tabla\_criba$ **do**
    $tabla\_criba[it] \leftarrow tabla\_criba[it]/base\_primos[i]$ {Dividimos}
    $factores[it][i] \leftarrow \neg factores[it][i]$ {Registramos el factor}
    $factores\_count[it][i] \leftarrow factores\_count[it][i] + 1$
    $it \leftarrow it + base\_primos[i]$
  end while
  $r2 \leftarrow (-r1) \mod base\_primos[i]$ {Calculamos segunda solución}
  $sol2 \leftarrow (r2 - x\_ini) \mod base\_primos[i]$
  if $sol1 \neq sol2$ **then**
    $it \leftarrow sol2$ {Cribamos con la segunda solución}
    while $it <$ longitud de $tabla\_criba$ **do**
      $tabla\_criba[it] \leftarrow tabla\_criba[it]/base\_primos[i]$
      $factores[it][i] \leftarrow \neg factores[it][i]$
      $factores\_count[it][i] \leftarrow factores\_count[it][i] + 1$
      $it \leftarrow it + base\_primos[i]$
    end while
  end if
end for
for $i \leftarrow 0$ **to** longitud de $tabla\_criba - 1$ **do**
  if $tabla\_criba[i] == 1$ **then**
    Añadir i a $x\_uniformes$
    Añadir $factores[i]$ a $factores\_uniformes$
    Añadir $factores\_count[i]$ a $factores\_uniformes\_count$
  end if
end for

Algorithm 8 Factorización Criba Cuadrática (2) - Resolución del sistema

```
1: if x_uniformes.size() > 0 then
2:   factores_uniformes_t ← traspuesta de factores_uniformes
3:   factores_uniformes_t ← eliminacionGaussiana(factores_uniformes_t)
4:   sols ← encuentraSoluciones(factores_uniformes_t)
5:   {Buscamos una solución que nos dé un resto cuadrático útil}
6:   for n_sol ← 0 to longitud de sols − 1 do
7:     sol ← sols[n_sol]
8:     if sol no es una solución trivial then
9:       Inicializar factores_r con ceros {Donde obtenemos el resto
         cuadrático resultante de la solución}
10:      for i ← 0 to longitud de sol − 1 do
11:        if sol[i] then
12:          for j ← 0 to longitud de base_primos − 1 do
13:            factores_r[j] += factores_uniformes_count[i][j]
14:          end for
15:        end if
16:      end for
17:      for j ← 0 to longitud de factores_r − 1 do
18:        factores_r[j] ← factores_r[j]/2 {Calculamos la raíz del resto
           cuadrático}
19:      end for
20:      Inicializar x_acum ← 1
21:      Inicializar r_acum ← 1
22:      for i ← 0 to longitud de sol − 1 do
23:        if sol[i] then
24:          x_actual ← x_ini + x_uniformes[i] {Obtenemos x usados}
25:          x_acum∗ = x_actual {Calculamos su producto}
26:        end if
27:      end for
28:      for j ← 0 to longitud de factores_r − 1 do
29:        for k ← 0 to factores_r[j] − 1 do
30:          r_acum ← r_acum ∗ base_primos[j]
31:        end for
32:      end for
33:      {Calculamos la diferencia x − y en (x − y)(x + y) ≡ 0 mód n}
34:      dif ← (x_acum − r_acum) mód n
35:      Calcular p ← gcd(dif, n)
36:      q ← n/p
37:      if p ≠ 1 y q ≠ 1 then
38:        return p
39:      end if
40:    end if
41:  end for
42: end if
43: return n
```

- *resuelveEcuacionCuadratica*, que tal y como el nombre indica, devuelve las soluciones de una ecuación de la forma $ax^2 + bx + c = 0$, con $a \neq 0$. Esta la necesitaremos para resolver ecuaciones de la forma $x^2 - (n - \phi(n) + 1)x + n = 0$, como dicta el algoritmo de Wiener.

A partir de estas funciones, usamos el esquema descrito en el apartado 5.2.1 para producir el código del ataque, que es el descrito en el algoritmo 9.

Algorithm 9 Ataque de Wiener

Require: e, n
Ensure: d, p, q (con toda seguridad si d es vulnerable)

```
1: cf = cocientesFraccionContinua(e, n)
2: for i ← 0 to longitud de cf − 1 do
3:    {Calculamos la reducida actual}
4:    d ← cf[i] · d1 + d2
5:    k ← cf[i] · k1 + k2
6:    d2 ← d1
7:    d1 ← d
8:    k2 ← k1
9:    k1 ← k
10:   {Si k = 0, pasamos a la siguiente reducida}
11:   if k ≠ 0 then
12:      aux ← ed − 1
13:      phi, r ← aux/k, aux %k
14:      if r = 0 then
15:         a ← 1
16:         b ← −n + phi − 1
17:         c = n
18:         p, q = resuelveEcuacionCuadratica(a, b, c)
19:         if existe solución and p · q = n then
20:            return (d, p, q)
21:         end if
22:      end if
23:   end if
24: end for
```

8.6. Experimentación y resultados

Una vez acabada la fase de implementación, pasaremos a poner a prueba las funcionalidades implementadas.

8.6.1. Generación de claves RSA

En primer lugar, vamos a estudiar el tamaño de las claves que se generan con las funcionalidades de generación de claves. Está claro que generar claves de cierto tamaño con primos aleatorios es una tarea sencilla, pues basta con asegurar que la suma de los dígitos que ocupan p y q sumen el tamaño deseado para n.

Por ello, nos centraremos en la generación de claves con primos robustos, pues la generación de los mismos es mucho menos trivial y genera módulos RSA de un tamaño menos consistente. La principal clave para generar estas claves de tamaño adecuado es elegir un tamaño apropiado para la generación de los primos r, s y t. Recordemos que estos valores deben ser, según [Gor85], de tamaño "algo menor" que $n/2$ en el caso de r y s, y t algo menor que r.

Tras varias pruebas, hemos decidido que para la generación de una clave de longitud en bits b, s tendrá un tamaño de $tam_s = b - log_2(b)/2 - 1$ y t tendrá un tamaño de $tam_t = tam_s - log_2(tam_s)/2 - 1$. Esto produce claves muy cercanas al número de bits deseado, pero es susceptible a errores considerables para valores bajos de b. Si no dar con un primo robusto de un tamaño exacto fuera un problema, igualmente podemos repetir el proceso tantas veces como queramos hasta llegar a un primo robusto del tamaño exacto. En este trabajo no lo consideraremos necesario.

8.6.2. Funcionamiento de cifrado/descifrado RSA

Para dar fe de que nuestra clave RSA funciona correctamente, en este apartado vamos a emplearlo para cifrar y descifrar el mensaje $m = 42$. Para ello, accedemos al programa, generamos una clave de 2047 bits y ciframos con ella el mensaje propuesto. Finalmente, desciframos el criptograma resultante del cifrado y comprobamos que $D(E(m)) = m$. Esta prueba puede observarse en la figura 8.3.

8.6.3. Eficiencia de los ataques

En esta sección, vamos a ejecutar los ataques sobre el RSA programados con el fin de estudiar su eficiencia. Este proceso lo haremos tanto con claves aleatorias como robustas para observar el efecto que tienen estas en el éxito de los ataques. Los resultados obtenidos se ubican en los cuadros 8.6.3 y 8.6.3.

En primer lugar, vamos a fijarnos en los resultados de los tres primeros métodos de factorización. Vemos claramente cómo estos algoritmos consiguen factorizar de manera bastante rápida los números de tamaño menor a 60 bits. Sin embargo, el ataque de Fermat se queda estancado sobre los 60

```
----Clave generada----
n = 830794649243332514669467193921418583670083901744173198377206724857054308682968214719973124787047582264351099764697878751808325187233947379076280918990728161418979541613911334303498959788966616302728111413692079663841825112216294118207356866758052982395232392562254735100871106780980574064972129693632925317555590205887112713186571495836893804921707866996363162470900371055748549181650637250171941427035958165394620230171986655205365482287406193330215636051128872029014391188542748776924231523815961919943895914604473951589384854368360239120187571561606639586195135036635368304871320438051877214851533289295696890394390e = 65537
d = 356304762291260615985088033012638847234616296539717656999681850184738780446956491922643462752178866818800642020669794586381855434392151851600923185055880489544842238698967309875747204270547148457239084833765114794866414664127230997421997008489812770154542925153344360955613140971196455430129807054486687873872808252444698731890279352648342670237985687937248642764157304014764925315728607640620990801605939508992560065696008558624438895394040438565917854227984843356750793656450948222981012481448241195686944689338703228331426037830557993343728743695154386243905311781614274001386517354903224429292055164661471033923370
tamanio n = 2047
========== Menu ===========
1. Crear clave RSA
2. Cifrar mensaje
3. Descifrar mensaje
4. Atacar clave
5. Salir
===========================
Seleccione una opción: 2
Introduce el numero a cifrar: 42
Mensaje = 42
Cifrado = 700771849868396299547442939571645927722245593327091253798652644264545458211909468635989753981380683813627383946343305109946111172706031810557565274298832033675789307838961110282847663226875416548888803076155639139704216459695694358981405091663840105565103342361212785002188688007953326643711339942002826125027869432450498021847798856806282045062585407875830592077010690881650355166553025325523317246400688839512767230078759185208942268088406224842733706585588282054472023026164863329422995473285772038997835851595513744545904405190789751495610455481791618148728268149879852220548025525947964221047439428302324877333
========== Menu ===========
1. Crear clave RSA
2. Cifrar mensaje
3. Descifrar mensaje
4. Atacar clave
5. Salir
===========================
Seleccione una opción: 3
Introduce el numero a descifrar: 700771849868396299547442939571645927722245593327091253798652644264545458211909468635989753981380683813627383946343305109946111172706031810557565274298832033675789307838961110282847663226875416548888803076155639139704216459695694358981405091663840105565103342361212785002188688007953326643711339942002826125027869432450498021847798856806282045062585407875830592077010690881650355166553025325523317246400688839512767230078759185208942268088406224842733706585588282054472023026164863329422995473285772038997835851595513744545904405190789751495610455481791618148728268149879852220548025525947964221047439428302324877333Cifrado = 700771849868396299547442939571645927722245593327091253798652644264545458211909468635989753981380683813627383946343305109946111172706031810557565274298832033675789307838961110282847663226875416548888803076155639139704216459695694358981405091663840105565103342361212785002188688007953326643711339942002826125027869432450498021847798856806282045062585407875830592077010690881650355166553025325523317246400688839512767230078759185208942268088406224842733706585588282054472023026164863329422995473285772038997835851595513744545904405190789751495610455481791618148728268149879852220548025525947964221047439428302324877333Mensaje original = 42
```

Figura 8.3: Cifrado y descifrado de un mensaje con una clave RSA de 2047 bits.

Algoritmo	**30 bits**	**60 bits**	**90 bits**	**120 bits**
Fermat	$0,1\ ms$	$40\ ms$	-	-
Kraitchik	$0,051\ ms$	$0,01\ ms$	$1,3\ ms$	$280\ ms$
ρ de Pollard	$0,01\ ms$	$3\ ms$	$1,39\ s$	$5,58\ s$
$p-1$ de Pollard	$0,5\ ms$	$2\ ms$	$7\ ms$	$3,65\ s$
Criba Cuadrática	$0,07\ s$	$490\ ms$	$11,26\ s$	$348,7\ s$

Cuadro 8.1: Tiempos de factorización de los algoritmos en función del tamaño de entrada con claves aleatorias. Las entradas vacías significan que el algoritmo no ha conseguido factorizar el número en dicho caso.

Algoritmo	**30 bits**	**60 bits**	**90 bits**	**120 bits**
Fermat	$0,2\ ms$	$5,21\ s$	$5,08\ s$	-
Kraitchik	$0,1\ ms$	$6,73\ s$	$10\ ms$	-
ρ de Pollard	$0,1\ ms$	$29\ s$	$646\ ms$	-
$p-1$ de Pollard	$0,1\ ms$	$2\ ms$	$60\ ms$	$9,26\ s$
Criba Cuadrática	$7\ ms$	$353\ ms$	$4,02\ s$	$342,5\ s$

Cuadro 8.2: Tiempos de factorización de los algoritmos en función del tamaño de entrada con claves robustas. Las entradas vacías significan que el algoritmo no ha conseguido factorizar el número en dicho caso.

bits y los de Kraitchik y ρ de Pollard sobre los 90 para el caso de la clave robusta, dejando de ser útiles para entradas más grandes si estos no cumplen propiedades deseables, como que el número se descomponga en primos muy cercanos.

Por otro lado, tenemos $p-1$ de Pollard. Para curarnos en salud, elegiremos $K = \sqrt{n}$, pues es el valor máximo que puede tener p y no trabajaremos con números muy grandes. En consecuencia, si dejamos al algoritmo trabajar el tiempo suficiente, llegaremos a la factorización del número. Los resultados en este caso son los esperados: cuando empleamos como clave un primo robusto, este presenta al menos un factor primo de gran tamaño, por lo que el método tendrá que recorrer todos los primos anteriores a él para factorizar el número.

Finalmente, llegamos a la criba cuadrática. Para este algoritmo, el autor recomienda emplear $B \approx e^{\frac{1}{2}\sqrt{ln(n)\cdot ln(ln(n))}}$ [Pom96]. Sin embargo, dada nuestra implementación, ha resultado mucho más beneficioso hacer este ajuste en función de los restos cuadráticos que el algoritmo encontraba. Si B es muy pequeño para el número a factorizar y se encuentran muy pocos restos cuadráticos haciendo la criba, entonces aumentamos el valor del mismo, mientras que si notamos que se obtienen muchos restos pero tenemos demasiados primos en la base de primos, podemos considerar bajar dicho valor. Valores típicos para la criba eran $B = 5000$ para 60 dígitos, $B = 10000$ para

90 y $B = 50000$ en el caso de 100 dígitos.

En cuanto al número de valores sobre los que realizar la criba, los valores suelen rondar el orden de los millones. Para no tener que preocuparnos por él, hemos incluido en la implementación que la criba pare cuando encuentre tantos restos cuadráticos como valores tengamos en nuestra base de primos, más un margen de 50 restos más. Así, tendremos un buen conjunto de soluciones posibles con las que crear la congruencia final, y si una congruencia falla, probamos con otra solución.

Este algoritmo, a diferencia del resto, parece no haber sido muy afectado por la condición de que el número a factorizar sea primo robusto. Sin embargo, los tiempos que emplea para factorizar con la implementación actual son escandalosos. Por suerte, el proceso de cribado puede realizarse de forma paralela, por lo que podría conseguirse un mejor rendimiento empleando más núcleos de procesamiento.

En conclusión, el algoritmo $p - 1$ de Pollard es el que mejores tiempos ha marcado en las pruebas que hemos realizado. Sin embargo, este es un algoritmo de factorización que depende en gran parte del número a factorizar, pues su eficiencia viene dada por el mayor factor primo del número a factorizar menos 1. Por contra, la criba cuadrática es un algoritmo mucho más pesado, pero funcional para números de mayor tamaño (probablemente a partir de 120 bits) y no tiene una dependencia tan fuerte de las condiciones concretas del número a factorizar.

Igualmente, nos quedamos muy lejos de poder llegar a factorizar un módulo RSA. De hecho, aún queda un largo camino para que eso pase, al menos por la vía clásica. El récord mundial de factorización actualmente es una clave RSA de 250 dígitos decimales mediante la criba general del cuerpo de números [Knu20]. Esto equivale a un número de 829 bits, lo cual no es ni la mitad de una clave RSA actual.

Funcionamiento del ataque de Wiener

En esta sección, vamos a mostrar los resultados obtenidos por el ataque de Wiener. Este resulta ser, en los casos donde funciona, tremendamente eficaz, pudiendo romper claves RSA de tamaño prácticamente arbitrario. Esto concuerda con la teoría, pues las operaciones que componen el ataque son calcular la representación de una fracción como fracción continua (lo cual tiene la misma eficiencia que el Algoritmo de Euclides) y calcular las soluciones de una ecuación de segundo grado (lo cual es también muy rápido). En la figura 8.4, mostramos un ejemplo de estas ejecuciones con un módulo de cifra n de 10.000 bits.

Por suerte o por desgracia, este ataque no suele funcionar en claves RSA

Figura 8.4: Aplicación del ataque de Wiener a una clave RSA vulnerable.

elegidas aleatoriamente. Sin embargo, no es mala idea comprobar si una clave RSA es susceptible a este ataque antes de usarla.

Capítulo 9

Implementación de algoritmos cuánticos

Una vez realizado el estudio e implementación de los ataques clásicos, en este capítulo vamos a tratar sobre la implementación de los algoritmos cuánticos.

Comenzaremos hablando sobre los recursos y materiales que hemos empleado para esta parte, los cuales son completamente diferentes a los empleados en los ataques clásicos. Esto se debe a las necesidades específicas que tiene la programación y ejecución de circuitos cuánticos. Programar estas tareas en C++ resulta bastante costoso, por lo que emplearemos un entorno de trabajo más simple y amigable.

Una vez familiarizados con las herramientas de esta nueva parte, realizaremos una breve descripción de la implementación de las diferentes partes del algoritmo de Shor. Además, experimentaremos con dicho circuito y veremos cómo factorizar algunos números pequeños, atendiendo a la eficiencia del algoritmo.

Finalmente, implementaremos simulaciones de los protocolos de intercambio de claves mediante canales cuánticos vistos en el capítulo 7. Haciendo uso de estas simulaciones, experimentaremos los efectos que provoca un espía en el protocolo y analizaremos su seguridad.

9.1. Recursos empleados

En este caso, nuestro entorno de trabajo serán notebooks de Jupyter sobre un entorno de Anaconda. Por tanto, nuestro lenguaje de programación será Python. Este lenguaje resulta ideal para la tarea propuesta, pues dispone de las librerías necesarias para lidiar con algoritmos cuánticos y permite

el envío de tareas a ordenadores cuánticos reales de manera sencilla.

De la misma forma que con la implementación clásica, también se emplea GitHub para el control de versiones del proyecto. El código relativo a esta parte estará almacenado en la carpeta *notebooks* del proyecto. El repositorio se puede encontrar en el siguiente enlace: https://github.com/jorgelerre/tfg_rsa.

9.1.1. Qiskit

Para realizar las implementaciones de algoritmos cuánticos, haremos uso de la librería Qiskit. Esta permite la creación y ejecución de algoritmos cuánticos definiéndolos bajo el modelo de circuito cuántico, tal y como hemos considerado a lo largo de este trabajo. Podemos encontrar la documentación de la librería en [Qis24]. A modo de ilustración de las principales funciones de Qiskit, vamos a mostrar cómo se implementaría el circuito de teleportación cuántica visto en el capítulo 3.

Para crear un circuito cuántico, podemos definir una serie de registros cuánticos y clásicos empleando los tipos de datos *QuantumRegister* y *ClassicalRegister*, respectivamente. Los registros cuánticos se corresponderán con nuestros qubits, mientras que los registros clásicos serán usados para almacenar las mediciones de los qubits. Cuando tengamos nuestros registros preparados, podremos crear un circuito cuántico que contemple dichos qubits con la clase *QuantumCircuit*.

```
1 qubit_tp = QuantumRegister(1, name='qubit_tp')
2 qubits_epr = QuantumRegister(2, name='par_epr')
3 bits_clasicos = ClassicalRegister(1, name='medicion')
4 circuito = QuantumCircuit(qubit_tp, qubits_epr,
     bits_clasicos)
```

Listing 9.1: Creación de un circuito cuántico con tres qubits y un registro clásico.

Una vez tenemos inicializado nuestro circuito con los registros apropiados, podemos aplicar puertas cuánticas sobre el mismo empleando los métodos de la clase *QuantumCircuit* creados a tal efecto. Dichas puertas se irán incluyendo en el circuito de izquierda a derecha. Si la puerta es de un qubit, solo necesitaremos especificar como parámetro el número del qubit sobre el que aplicarla. Si tenemos una puerta controlada de dos o más qubits, escribiremos siempre los qubits de control primero, seguidos del qubit objetivo.

Qiskit incluye todas las puertas que hemos visto en el trabajo. Además, contempla puertas paramétricas básicas, las cuales permiten girar un qubit en la esfera de Bloch cualquier ángulo sobre los ejes X, Y o Z. De esta forma, empleando únicamente tres puertas, se permite la creación de cualquier

puerta de un solo qubit.

Para aplicar una medición, emplearemos el método *measure* sobre el circuito que hemos creado. Este espera como parámetros una lista de qubits a medir y una lista de bits clásicos donde almacenar el valor medido. Cuando ejecutemos el circuito, la salida que obtendremos será esencialmente el resultado de las mediciones que hagamos.

Cuando hayamos incluido las puertas que conforman el circuito, podremos visualizarlo con el método *draw*.

```
circuito.x(0)
circuito.h(2)
circuito.cx(2,1)
circuito.cx(0,1)
circuito.h(0)
circuito.barrier()
circuito.measure([0,1],[0,0])
circuito.barrier()
circuito.cx(1,2)
circuito.cz(0,2)
circuito.measure(2,0)
```

Listing 9.2: Adición de puertas a un circuito cuántico.

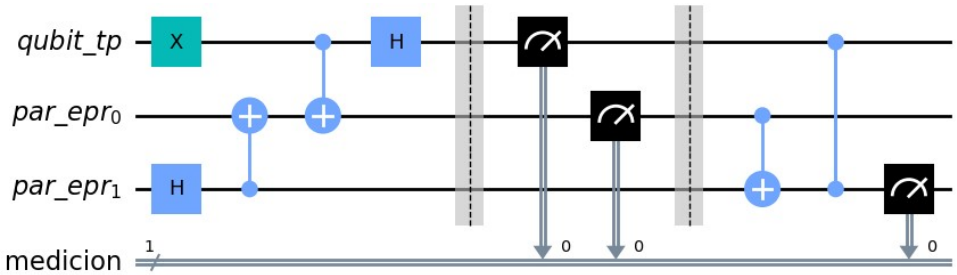

Figura 9.1: Circuito de teleportación cuántica en Qiskit, visualizado con el método *circuito.draw('mpl')*.

Tras construir el circuito, vamos a ejecutarlo. En este punto, tenemos dos opciones: ejecutar el circuito en nuestro propio ordenador mediante simuladores o mandar la tarea a un ordenador cuántico real. En ambos casos, los pasos a seguir resultan bastante similares:

1. Seleccionar un *backend*: elegimos la máquina o el simulador con el cuál realizar la ejecución. Si usamos simulaciones, en este paso es muy recomendable usar una GPU para que los tiempos de ejecución sean asumibles.

2. Transpilar el circuito: este es el equivalente en computación cuántica a la compilación en computación clásica. Para poder ejecutar el circuito, es necesaria su traducción a instrucciones que pueda ejecutar a bajo nivel el *backend* escogido.

3. Ejecutar el circuito: una vez compilado, ejecutamos el circuito cierto número de veces (por defecto, 1.000). Este nos devolverá un objeto con los resultados y diferentes estadísticas de la ejecución.

4. Mostrar los resultados: una representación típica de los resultados de un circuito cuántico es un histograma de los resultados obtenidos en diferentes ejecuciones del circuito.

```
simulator = Aer.get_backend('aer_simulator', method =
    'statevector', device = 'gpu')
qc_suma_transpiled = transpile(circuito, simulator)

# Run and get counts
result = simulator.run(qc_suma_transpiled).result()
print(result)
counts = result.get_counts(qc_suma_transpiled)
plot_histogram(counts, title='Resultados')
```

Listing 9.3: Ejecución de un circuito cuántico en el simulador Aer.

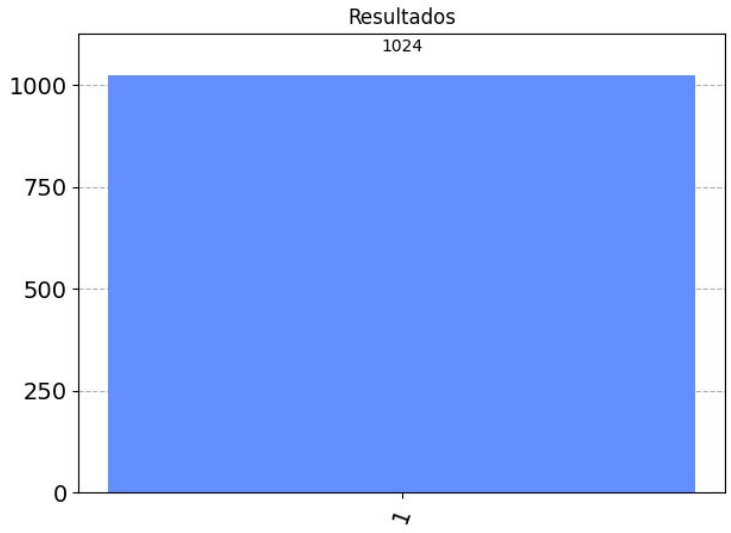

Figura 9.2: Resultados de 1.024 ejecuciones del circuito de teleportación cuántica en el simulador. El resultado es el esperado, pues el qubit teleportado tenía el valor $|1\rangle$.

9.1.2. Ordenadores cuánticos

Para acceder a ordenadores cuánticos reales y ejecutar nuestros circuitos, vamos a emplear la plataforma *IBM Quantum* [Qua24]. En esta, tenemos a nuestra disposición distintos ordenadores cuánticos. Para nuestras pruebas, emplearemos tres: Brisbane, Osaka y Sherbrooke.

Cada ordenador cuántico presenta una disposición de qubits concreta. A esta manera de conectar los qubits entre sí se le llama *topología*. Se representa como un grafo, donde los nodos se corresponden con los qubits del ordenador y las aristas representan los qubits que pueden interaccionar directamente entre sí. De esta forma, solo podremos ejecutar puertas de dos qubits sobre qubits contiguos en el grafo. Los tres ordenadores que utilizaremos presentan la misma topología, pues su procesador cuántico es el mismo (Eagle R3). Esta topología se puede ver en la figura 9.3

Por otro lado, los ordenadores cuánticos pueden implementarse con diferentes *conjuntos de puertas* nativos. Cada tipo de puerta poseerá cierta precisión y tasa de error. En nuestro caso, los tres ordenadores que hemos escogido se basan en las mismas puertas básicas:

- Puerta ECR: Permite crear entrelazamiento entre dos qubits.
- Puerta ID: Se corresponde con la matriz identidad: al aplicarla sobre un qubit, lo deja igual.
- Puerta RZ: Es una puerta parametrizable, la cual aplica un cambio de fase arbitrario sobre un qubit.
- Puerta SX: Es la llamada puerta $\sqrt{X}$, que gira el estado de un qubit 90^{o} en la esfera de Bloch.
- Puerta X.

Nosotros, por suerte, no tendremos que considerar a la hora de implementar circuitos estas características. La adaptación de los circuitos a las arquitecturas específicas de los ordenadores se realiza automáticamente mediante el proceso de transpilación, anteriormente mencionado.

Otra característica clave de un ordenador cuántico son los tiempos que puede mantener un qubit en cierto estado. Estos son los llamados *tiempos de decoherencia cuántica*. Esta decoherencia se debe a la interacción del sistema del ordenador cuántico con el exterior, la cual es muy difícil de evitar. Podemos distinguir dos tiempos:

- *T1 o tiempo de relajación*: es el tiempo que tarda un qubit en decaer desde el estado excitado $|1\rangle$ al estado base $|0\rangle$. Esto se debe a la disipación de energía del qubit con el entorno.

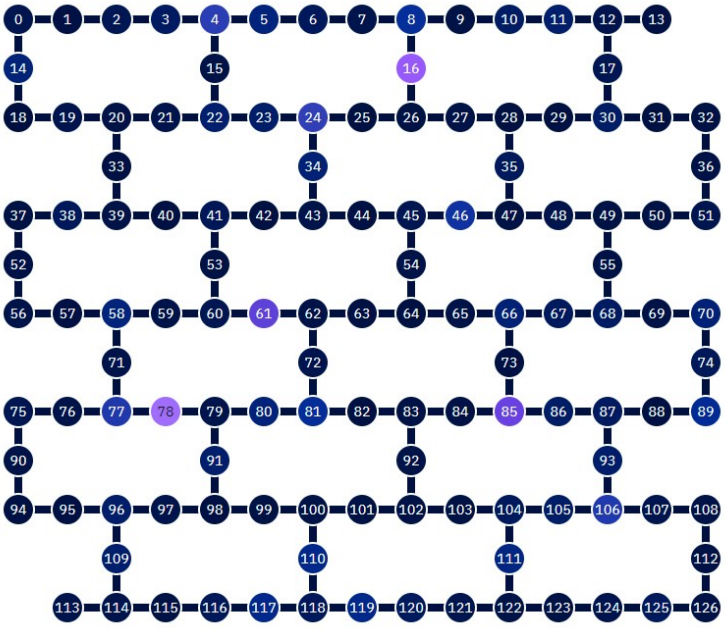

Figura 9.3: Topología de los ordenador cuánticos implementados con Eagle R3. Vemos que la conectividad entre qubits no es plena, sino que se disponen en forma de malla para minimizar la interacción y hacer que los tiempos de decoherencia sean más altos. [Qua24]

- *T2 o tiempo de coherencia*: es el tiempo que tarda un estado en superposición en perder la correlación de las fases de $|0\rangle$ y $|1\rangle$.

Debido a esto, no podremos ejecutar en un ordenador cuántico real circuitos demasiado largos (mientras no se disponga de corrección cuántica de errores), pues llegará un punto en el que los qubits se degraden y obtengamos resultados incorrectos.

En [Wac+21] se señalan las principales métricas que definen las prestaciones de un ordenador cuántico. Estas son:

- La escala o número de qubits, pues este determina la cantidad de información que puede ser codificada en el ordenador. Puede ser un factor limitante en algoritmos que requieran mucha memoria.
- El volumen cuántico: es una métrica integral de las capacidades de un ordenador cuántico. Este se define como 2^x, donde x es el mayor número tal que un circuito de x qubits de entrada con x capas de puertas de 2 qubits que tenga una tasa de error razonable ($< 33\,\%$). Esta medida refleja las capacidades de resolución de problemas de un ordenador cuántico de manera bastante concisa, teniendo en cuenta

tanto la precisión de sus operaciones como los tiempos de decoherencia. Actualmente, 128 y 256 son valores típicos de esta métrica para los dispositivos de IBM.

- Las operaciones a nivel de capa de circuito por segundo (CLOPS): esta métrica refleja cuántos circuitos del tamaño del volumen cuántico pueden ejecutarse en el ordenador cuántico por segundo. Esta considera tanto el tiempo de ejecución del circuito como los tiempos de preparación y de guardado de las medidas realizadas.

Este último año, IBM ha introducido una nueva métrica para caracterizar mejor el rendimiento de un ordenador cuántico [Qua23]. Esta medida es el *Error Per Layered Gate* (EPLG), que expresa la probabilidad de error de una puerta al aplicarla de manera conjunta con puertas sobre el resto de qubits.

Característica	**Brisbone**	**Osaka**	**Sherbrooke**
Número de Qubits	127	127	127
Volumen Cuántico	128	128	128
CLOPS	5K	5K	5K
EPLG	2.1 %	2.2 %	2.7 %
T_1 mediana (μs)	229.69	252.89	272.79
T_2 mediana (μs)	135.56	175.92	133.28

Cuadro 9.1: Comparación de características de los ordenadores cuánticos de IBM en el momento de su uso en este trabajo.

Volviendo a nuestro ejemplo, vamos a ejecutar el algoritmo de teleportación en un circuito cuántico real. El proceso para ello es muy similar al anterior: usamos una clave de sesión de IBMQ para identificarnos, seleccionamos el ordenador cuántico donde queramos ejecutar el circuito, transpilamos y mandamos el trabajo a la cola del ordenador. Los resultados los podremos conseguir posteriormente en la página de IBMQ.

```
# Inicio de sesion
IBMQ.save_account(clave_sesion)
IBMQ.load_account()
# Indicamos el dispositivo a usar
provider = IBMQ.get_provider(hub='ibm-q')
backend = provider.get_backend('ibm_osaka')
# Transpilamos el circuito al ordenador cuantico
transpiled_qc = transpile(circuito, backend)
# Mandamos el trabajo a la cola
job = execute(transpiled_qc, backend, shots=1000)
```

Listing 9.4: Ejecución de un circuito cuántico en IBM Osaka.

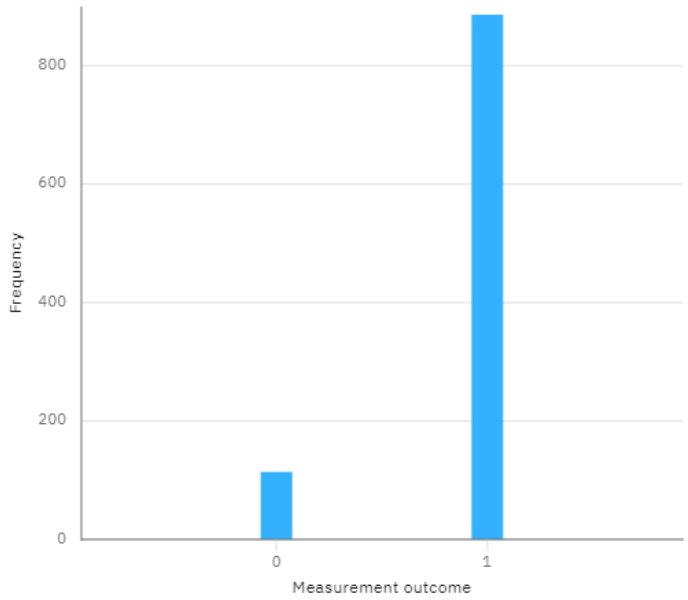

Figura 9.4: Resultados de 1.000 ejecuciones del circuito de teleportación en el ordenador IBM Osaka.

A diferencia de con las simulaciones, al emplear ordenadores reales vemos que aparece cierto porcentaje de ruido en los resultados, debido a los diferentes factores que hemos comentado. En este caso, el resultado (figura 9.4) no ha sido el esperado en 114 de las 1000 ejecuciones del circuito, lo cuál supone un 11,4 % de error para un circuito bastante pequeño.

9.2. Algoritmo de Shor

La implementación de esta parte se encuentra en el código adjunto al trabajo, en el fichero *ShorAlgorithm.ipynb*.

Para implementar el algoritmo de Shor en Qiskit, hemos ido construyendo los diferentes subcircuitos que lo forman (véanse los apartados 6.1.1 y 6.1.2) y probando su funcionamiento de manera individual. En la figura 9.5 se muestra el árbol de dependencias entre los circuitos implementados. En él se puede observar que hemos implementado dos versiones del algoritmo de Shor: una primera donde se usa la suposición de que el periodo de la función buscada es potencia de 2, y la versión general por otro lado.

Para dar una implementación más general, cada subcircuito tiene asociada una función tal que, dados ciertos parámetros, devuelve el circuito

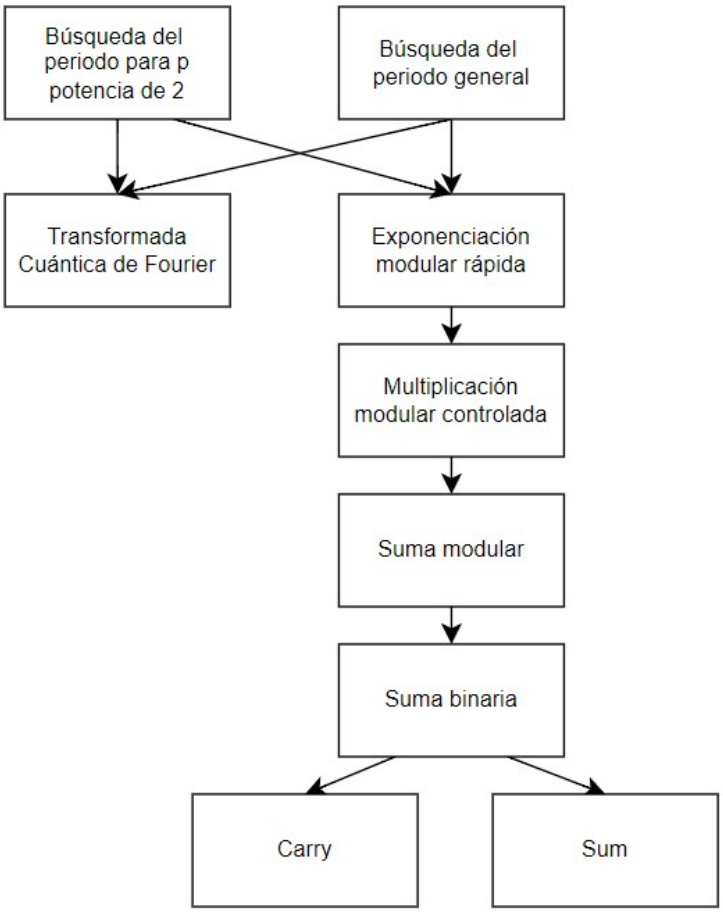

Figura 9.5: Árbol de dependencias entre subcircuitos del algoritmo de Shor.

correspondiente. De esta forma, cuando se realice una llamada al algoritmo de Shor, este puede generar el circuito adecuado para factorizar el número que se le pida.

- La suma binaria se implementa en la función *createAdder*, que recibe como parámetro el tamaño en qubits n de las entradas a sumar.
  - Entrada: el sumando A en los primeros n qubits, el sumando B en los siguientes n, y $n-1$ qubits auxiliares con valor $|0\rangle$.
  - Salida: el sumando A (sin modificar), $A+B$ en los siguiente $n+1$ qubits, y el resto de qubits auxiliares con valor $|0\rangle$.
- La suma modular, generada por la función *createModAdderVedral* (nombrada así en honor al autor original del circuito), recibe el módulo M con el que se trabaja, además del número de qubits de los valores de entrada n.

  - Entrada: el sumando A en los primeros n qubits, el sumando B en los siguientes n, y $2n+2$ qubits auxiliares con valor $|0\rangle$.
  - Salida: el sumando A en los primeros n qubits, la suma $A+B$ en los siguientes n, y $2n+2$ qubits auxiliares con valor $|0\rangle$.

- La multiplicación modular controlada podrá crearse llamando a la función *createMult*. A esta habrá que especificarle los valores n y M anteriores, además del número y por el que multiplicar la entrada.

  - Entrada: el qubit de control, seguido de los n qubits correspondientes al factor A en los siguientes n qubits y $3n+2$ qubits auxiliares con valor $|0\rangle$.
  - Salida: el qubit de control y el factor A (sin modificaciones), el producto resultante $y \cdot A \ (mod\ M)$ en los siguientes n qubits, y $2n+2$ qubits auxiliares con valor $|0\rangle$.

- La exponenciación modular rápida está programada en la función *createExpMod*. Esta recibe como parámetros n_{in}, que es el número de qubits que codifican el exponente de entrada a la función; n_{out}, que define el número de qubits de la salida; y, la base a usar en la exponenciación; y M, el módulo sobre el que se opera.

  - Entrada: los n_{in} qubits del exponente e y $5n_{out}+2$ qubits con valor $|0\rangle$.
  - Salida: los qubits del exponente sin modificaciones, $y^e \ (mod\ M)$ en los siguientes n_{out} y, finalmente, los $4n_{out}+2$ qubits restantes con valor $|0\rangle$.

- La transformada cuántica de Fourier se implementa en la función QFT, y recibe como parámetro el tamaño del registro sobre el que aplicar la transformación.

  - Entrada: n qubits con valor $|A\rangle$.
  - Salida: n qubits con valor $QFT(|A\rangle)$.

- El circuito de cálculo del periodo para periodos potencias de 2 (generado por la función *busqueda_periodo_potencia2*) recibe como parámetros num (el número que se desea factorizar, y que hará de módulo en el circuito de exponenciación subyacente) y la base b a emplear en el mismo.

  - Entrada: $6n+2$ qubits con valor $|0\rangle$, donde $n = \lceil log_2(num) \rceil$, y n bits clásicos.
  - Salida: un múltiplo de $\frac{2^n}{p}$, medido en los registros clásicos.

- El circuito de cálculo del periodo general (generado por la función *busqueda_periodo*) es exactamente igual al anterior, con la pequeña diferencia de que emplea n qubits y bits más para trabajar en un dominio más grande de valores y minimizar el error, tal y como comentamos en el capítulo 6.

Una vez definidas las anteriores funciones, podemos programar el Algoritmo de Shor en sus dos versiones.

```
def Shor_potencia2(n):
  M = 2**(math.ceil(math.log2(n)))
  # Buscamos una base para la funcion periodica
  for base in range(2,n):
    # Si gcd(base,n) fuera mayor que 1, realmente ya
    # habriamos factorizado el numero
    # Sin embargo, esto no nos interesa ahora
    if math.gcd(base, n) == 1:
      # Calculamos el periodo de la funcion
      # Ejecutamos el circuito en un simulador cuantico
      simulator = Aer.get_backend('aer_simulator',
          device = 'gpu')
      find_period_transpiled =
          transpile(busqueda_periodo_potencia2(15,
          base), simulator)
      result = simulator.run(find_period_transpiled,
          shots=1000).result()
      counts = result.get_counts(find_period_transpiled)

      for c in counts:
        # Convertimos el string en binario a int
        res = int(c, 2)
        if res != 0:
          posible_periodo = M // res
          if posible_periodo % 2 == 0:
            posible_raiz = base**(posible_periodo//2) % n
            if (posible_raiz*posible_raiz) % n == 1
                and posible_raiz != 1
                and posible_raiz != n - 1:
              # Si encontramos una raiz, tenemos la
                  factorizacion de n
              p = math.gcd(posible_raiz - 1, n)
              q = n // p
              return p, q
```

Listing 9.5: Algoritmo de Shor para periodos potencia de 2.

Para el algoritmo de Shor general, recordemos que necesitamos lidiar con fracciones continuas. Para ello, emplearemos la librería *sympy*, que ofrece

métodos que nos ahorrarán algo de trabajo en este aspecto.

```
def Shor(n, factor_no_trivial = True):
  num_valores = 2**(2*math.ceil(math.log2(n)))
  # Buscamos una base para la funcion periodica
  for base in range(2,n):
    # Si gcd(base,n) fuera mayor que 1, realmente ya habriamos factorizado el numero
    # Sin embargo, esto no nos interesa ahora
    # y es un caso muy improbable con valores de n grandes
    if math.gcd(base, n) == 1:
      # Calculamos el periodo de la funcion
      # Ejecutamos el circuito en un simulador cuantico
      simulator = Aer.get_backend('aer_simulator', device = 'gpu')
      find_period_transpiled = transpile(busqueda_periodo(n, base), simulator)
      result = simulator.run(find_period_transpiled, shots=1000).result()
      counts = result.get_counts(find_period_transpiled)
      # Ordenamos los resultados por frecuencia
      counts_ordenado = dict(sorted(counts.items(), key=lambda item: item[1], reverse=True))

      for key in counts_ordenado.keys():
        #fraccion = Rational(2**(ceil(log2(n))),posible_periodo)
        posible_periodo = int(key, 2) # Convertimos el string en binario a int
        # Si la division es exacta, probamos la raiz resultante de p
        if posible_periodo != 0:
          # Calculamos las convergentes de posible periodo / 2^{n_bits}
          fraccion = Rational(posible_periodo,num_valores)
          fraccion_continua = continued_fraction(fraccion)

          for i in range(len(fraccion_continua)):
            a = list_to_frac(fraccion_continua[:i+1])
            posible_periodo = a.q
            if posible_periodo % 2 == 0:
              posible_raiz = base**(posible_periodo//2) % n
              if (posible_raiz*posible_raiz) % n == 1 and ((not factor_no_trivial) or (posible_raiz != 1 and posible_raiz !=
```

```
                    n - 1)):
33                  # Si encontramos una raiz, tenemos la
                        factorizacion de n
34                  p = math.gcd(posible_raiz - 1, n)
35                  q = n // p
36                  return p, q
```

Listing 9.6: Algoritmo de Shor general.

9.2.1. Ejecución del algoritmo de Shor con simuladores

En este apartado, vamos a ejecutar ambas versiones del algoritmo y a analizar las salidas obtenidas.

Shor limitado a potencias de 2

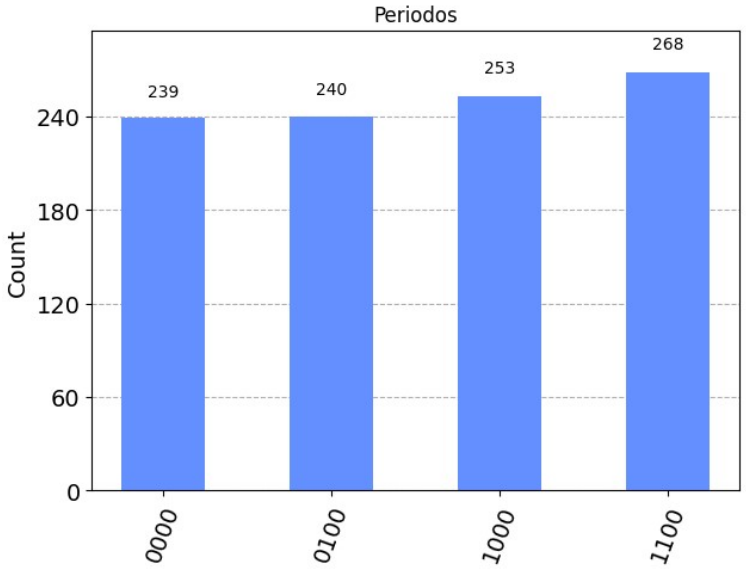

Figura 9.6: Resultados del circuito generado por la función *busqueda_periodo_potencia2(15, 2)*.

Comenzaremos con la versión limitada del mismo, factorizando el número 15. Realizando dicha ejecución, obtenemos efectivamente la factorización $15 = 5 \cdot 3$.

Si miramos qué ha pasado dentro del algoritmo, vemos que este crea el circuito *busqueda_periodo_potencia2(15, 2)*, y de este consigue el resultado de la figura 9.6. A partir de estos, sabiendo que todos responden a la expresión

$k\frac{M}{p}, k \in \mathbb{Z}$ y todos los resultados son múltiplos de 4, tenemos que p debe ser tal que $\frac{M}{p} = \frac{16}{p} = 4$. Despejando, sabemos que $p = 4$.

Posteriormente, el algoritmo usa este 4 para calcular una raíz cuadrada de la base (en este caso, 2). Con tal fin, calcula $2^{4/2} = 2^2 = 4$. Una vez obtenida la raíz cuadrada, podemos factorizar el número realizando $p = mcd(15, 4-1)$ y $q = n/p$, lo cual nos da efectivamente los factores 3 y 5 como solución.

Shor completo

En cuanto a la ejecución del algoritmo de Shor completo, nos encontramos con un problema: los simuladores no aceptan circuitos de más de 28 qubits, y si usamos $n = 4$ esta cantidad sube hasta los 30 qubits. Las máquinas reales sí aceptan valores mayores, pero presentan un grave problema de ruido, como veremos más adelante. Por tanto, lo máximo que podremos factorizar de esta manera es el número 6, pues es el único número compuesto que no es un cuadrado perfecto y está por debajo del 7.

Cuando ejecutamos el algoritmo en este caso, el único valor de $\mathbb{Z}_6$ que es coprimo con 6 es 5, por lo que este valor será usado como base. Así, el algoritmo de Shor llamará a la función *busqueda_periodo(6, 5)* y calculará el periodo de $f(x) = 5^x \ (mod\ 6)$. Como resulta que $5 \cdot 5 = 25 \equiv 1 \ (mod\ 6)$, el periodo es potencia de 2 y el circuito solo da dos salidas: 0 y $\frac{M}{2}$, tal y como se ve en 9.7. Como resultado, 5 es una raíz cuadrada de la unidad módulo 6, y se puede obtener un factor mediante el $mcd(5-1, 6) = 2$.

Para poder ver el comportamiento del algoritmo de Shor cuando la función no tiene un periodo potencia de 2, vamos a ejecutar el circuito *busqueda_periodo(7, 2)*. Como el número 7 es primo, sabemos por la teoría de números que el periodo de la función será 6. En la figura 9.8 vemos el resultado de la ejecución. Es claro el efecto de que el periodo no sea potencia de 2, ya que hace que las probabilidades de muchos estados no se anulen. Sin embargo, los máximos siguen dándose claramente en los estados $k\frac{M}{p} = k\frac{64}{6} \approx 10,6 \cdot k$.

9.2.2. Eficiencia de los circuitos

Una vez hemos comprobado el correcto funcionamiento de los circuitos, vamos a estudiar su eficiencia en función del número de puertas que utilizan y de su tiempo de ejecución.

Con este fin, en la parte final del notebook *ShorAlgorithm.ipynb* adjunto a la memoria se realizan ciertas simulaciones de los circuitos para diferentes tamaños de entrada. Concretamente, haremos tres simulaciones por cada

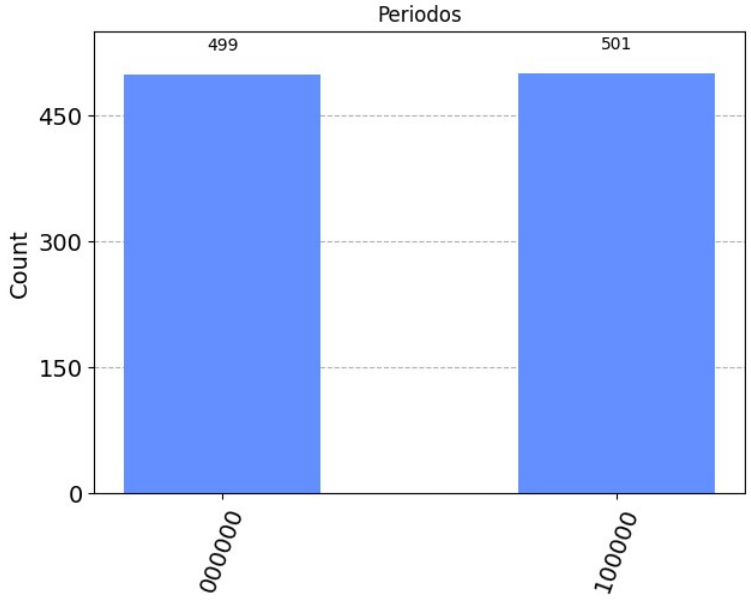

Figura 9.7: Resultados del circuito generado por la función *busqueda_periodo(6, 5)*.

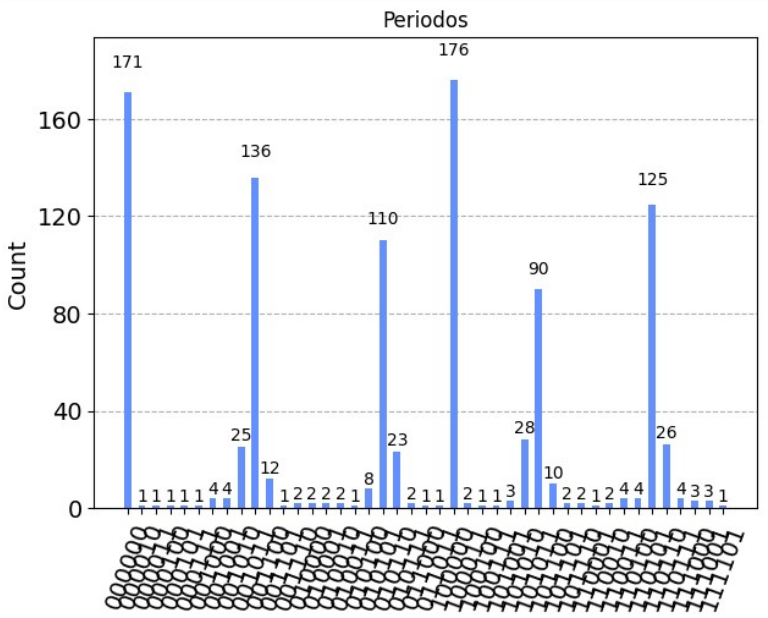

Figura 9.8: Resultados del circuito generado por la función *busqueda_periodo(7, 3)*.

número de qubits de entrada posibles, hasta alcanzar la barrera de los 28 qubits (a partir de la cual no podemos ejecutar el circuito en simuladores). En estas simulaciones, escogemos valores aleatorios para los parámetros de entrada de las funciones que generan los circuitos, y también colocamos a su entrada valores aleatorios. Después, ejecutamos los circuitos en el simulador y guardamos el tiempo y el número de puertas básicas del circuito (obtenido aplicando el método *transpile()* y la función *count_gates()*). Finalmente, consideramos la media de las tres simulaciones, obteniendo las figuras 9.9 y 9.10.

En las figuras, observamos que el tiempo y número de puertas se dispara para valores de entrada muy pequeños conforme el circuito se va volviendo más complejo, dándonos para la exponenciación modular rápida de tres qubits un circuito de casi 6.000 puertas. Sin embargo, el tiempo de ejecución parece bastante razonable: la ejecución más larga dura 0.08 segundos.

No obstante, cuando realizamos la ejecución de los circuitos de búsqueda del período, estos tiempos crecen notablemente. Para obtener el resultado en *busqueda_periodo_potencia2(15, 2)* se ha empleado un tiempo de más de 250 segundos, mientras que *busqueda_periodo(6,5)* y *busqueda_periodo(7,3)* tardan unos 30 segundos en obtener las salidas. Para entender este aumento tan drástico de tiempos de ejecución ante la inclusión de una cantidad proporcionalmente pequeña de puertas, debemos comprender que estamos ejecutando la tarea en un simulador. Cuando aplicamos una puerta de Hadamard sobre las entradas, estamos haciendo que se tenga que calcular el resultado del circuito para todas las entradas posibles, lo cual implica un aumento exponencial en la complejidad de los cálculos.

9.2.3. Uso de ordenadores cuánticos reales

En esta sección, vamos a probar los ordenadores cuánticos reales con los circuitos descritos.

Vista la gran cantidad de qubits y puertas cuánticas que presenta el circuito de búsqueda de un período, no tiene sentido intentar ejecutarlo en los ordenadores cuánticos actuales. Por el contrario, comenzaremos con un circuito muy básico: el sumador de dos qubits.

Si usamos el método decompose, este circuito presenta un total de 104 puertas cuánticas básicas. No obstante, este no es el número de puertas cuánticas que se usarán en el ordenador cuántico, pues no están representadas con las puertas nativas al mismo. Para obtener el número real, transpilamos el circuito al ordenador y volvemos a tomar la medida.

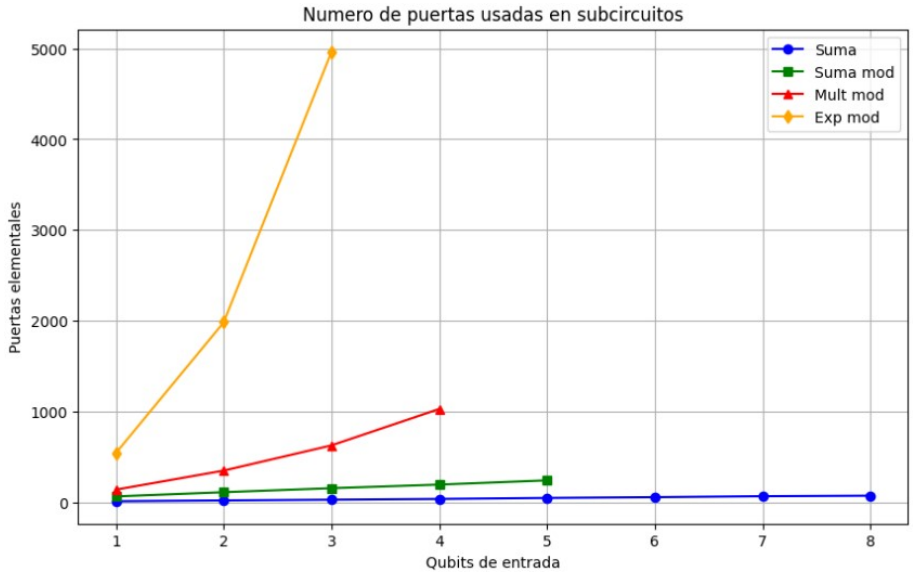

Figura 9.9: Número de puertas de los subcircuitos en función del número de qubits de entrada.

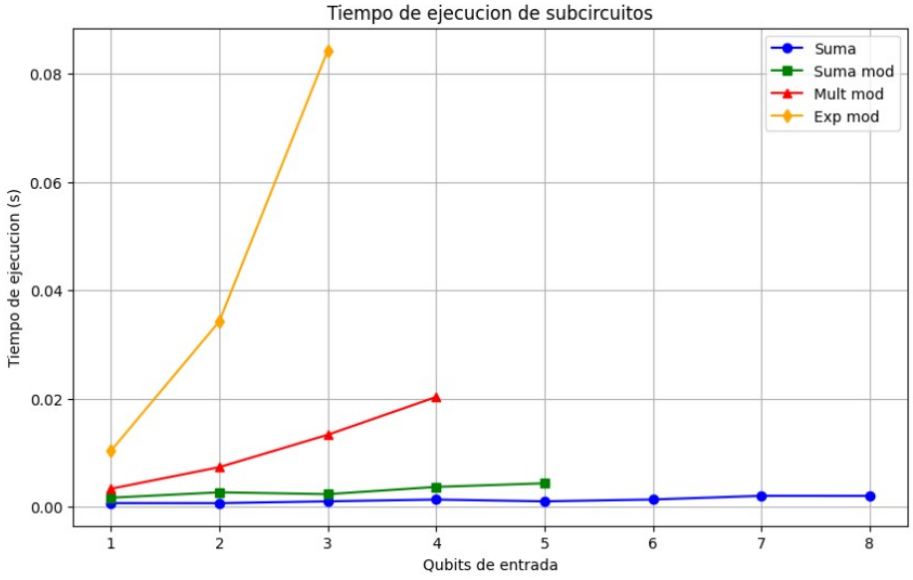

Figura 9.10: Tiempo de ejecución de los subcircuitos en función del número de qubits de entrada.

Entonces, comprobamos que el ordenador cuántico tendrá que ejecutar 436 puertas cuánticas nativas. Como este número cuadruplica el volumen cuántico de 128 de los ordenadores que tenemos a nuestra disposición, es muy probable que el error se descontrole y no obtengamos el resultado esperado.

En efecto, al ejecutar el algoritmo en *IBM-Osaka*, tenemos el resultado de la figura 9.11. Irónicamente, el resultado menos común es la solución de la suma, $|4\rangle$. Por tanto, los ordenadores cuánticos actuales están muy lejos de poder ejecutar un algoritmo tan complejo computacionalmente como el de Shor.

Una primera estimación de los recursos necesarios para ejecutar el circuito de búsqueda del período de una función sería disponer de un ordenador cuántico con un volumen cuántico de, aproximadamente, el número de puertas de dicho circuito. Este número vendrá en gran parte dado por el cuello de botella del algoritmo, que es el cálculo de la exponenciación modular rápida. En el caso de laboratorio de 3 qubits de entrada, ya necesitaríamos ejecutar 40.000 puertas, por lo que el volumen cuántico mínimo se correspondería con $2^{16} = 65536$.

No obstante, el algoritmo de Shor se ha conseguido ejecutar correctamente empleando ciertos trucos [Mon+16]:

- En vez de implementar las multiplicaciones modulares de forma genérica, se puede construir una tabla de verdad e implementar en el circuito la acción de esta tabla de verdad. Para números pequeños, este método es el que menos puertas usa, pero la eficiencia teórica del mismo es exponencial, por lo que no es factible para factorizar números grandes.

- Además, si podemos implementar el reseteo e inicialización de qubits dentro del circuito, podemos sustituir los n qubits de la entrada al algoritmo de exponenciación rápida por uno solo, el cual vaya adquiriendo el valor preciso de control sobre cada multiplicación.

Para un caso realista, donde el número a factorizar requiera 1024 qubits para su representación, el número de puertas (de las que consideramos "básicas" en este trabajo) es, según la cota teórica, de $702N^3 - 280N^2 + 4 = 702(1024)^3 - 280(1024)^2 + 4 = 7,534 \cdot 10^{11}$. Considerando la métrica del volumen cuántico y simplificando su interpretación a que sea el tamaño en puertas máximo del circuito para que los resultados sean coherentes, sería preciso con la implementación que hemos presentado que su valor sea de 2^40, lo cual es una barbaridad en comparación a los recursos actuales. Sin embargo, otras métricas deben ser también contempladas, como que los tiempos T_1 y T_2 sean lo suficientemente persistentes o que el número de qubits total sea el suficiente para ejecutar el circuito.

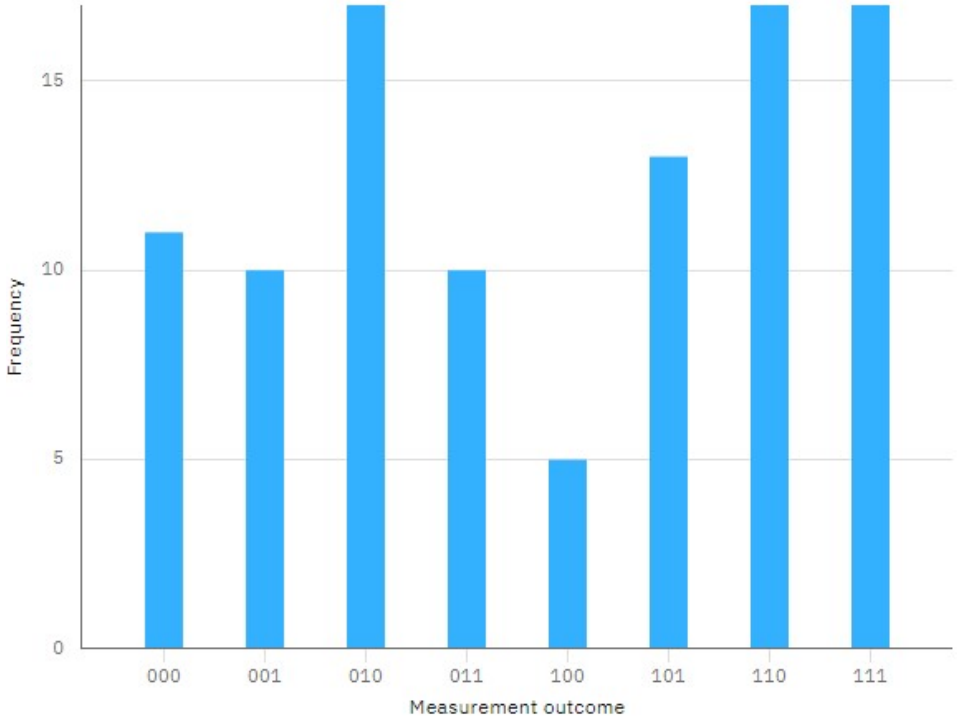

Figura 9.11: Resultados de la ejecución de la suma de qubits $|3\rangle + |1\rangle$.

9.3. Algoritmos de intercambio de clave

La implementación de esta parte se encuentra en el código adjunto al trabajo, en el fichero *QKD.ipynb*.

Esta vez, hemos separado la implementación de cada protocolo en dos funciones claramente diferenciadas:

- Una primera función generará un circuito, donde se simulará el intercambio de claves entre Alice y Bob usando el protocolo para un único bit. Como entradas, recibirá los valores de las elecciones aleatorias que realizan Alice y Bob, además las acciones de espionaje que puede ejecutar Eve entre medias. Estos parámetros concretos variarán entre protocolos, según las acciones que requiera cada uno.

- Una segunda función donde, dado un cierto número t y la presencia o no de un espía, ejecuta la función anterior t veces e implementa el protocolo correspondiente en su totalidad para t qubits transmitidos. Como salidas, devuelve la clave compartida y si se ha detectado algún espía, además de mostrar por pantalla ciertos datos internos del proceso si se indica la opción *verbose=True* en los parámetros.

Un detalle de implementación a tener en cuenta es que Qiskit solamente aplica mediciones sobre el eje Z. Por tanto, si queremos medir en otro eje (como en el eje X en los protocolos BB84 y B92, o en el eje 120^{o} en el caso de E91), tendremos que aplicar las puertas cuánticas correspondientes para cambiar de base al eje Z, aplicar la medición, y finalmente realizar la operación inversa para devolver el qubit al eje original.

Otro punto que cabe tratar es cómo actúa Eve cuando se especifica que esté presente en el protocolo.

- En BB84, mide todos los qubits en una base aleatoria (Z o X). Esto causará, como vimos anteriormente, que 1/4 de los bits que conforman la clave compartida de Alice y Bob no coincidan. Sin embargo, ella conocerá la mitad de los bits de esta.

- En B92, Eve aplicará la estrategia descrita en el capítulo 7.2. Igualmente, con suficientes qubits transmitidos, Alice y Bob deberían ser capaces de detectarla.

- En E91, Eve medirá todos los qubits entrelazados antes de que lleguen a Alice y Bob, de forma que los pares que recibirán no se comportarán siguiendo las leyes cuánticas.

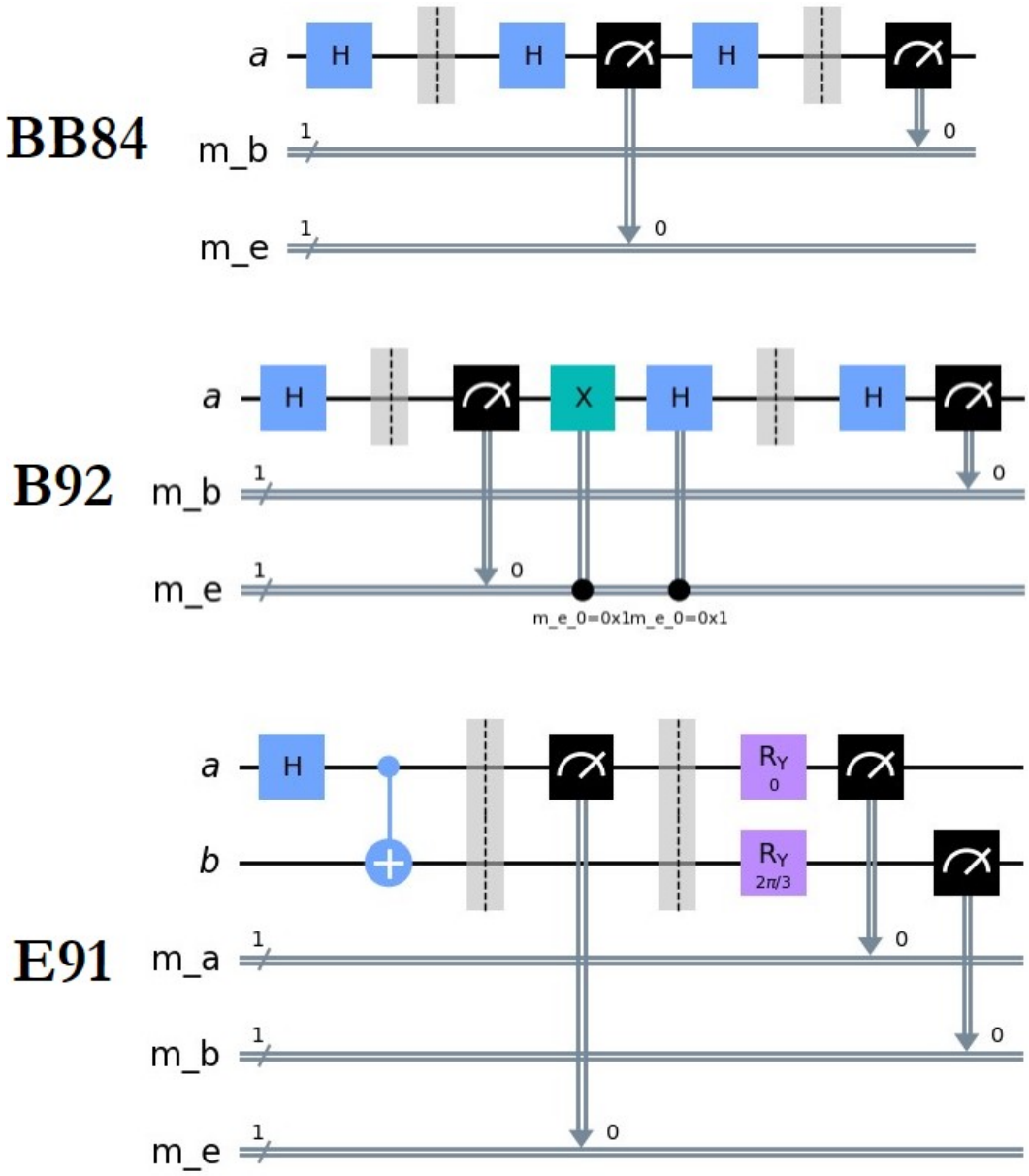

Figura 9.12: Circuitos de los protocolos de intercambio de claves con espionaje aplicado.

9.3.1. Experimentación y resultados

Por último, vamos a probar los circuitos implementados en el simulador y a estudiar su comportamiento.

En primer lugar, en la figura 9.13 vemos los resultados de una ejecución del protocolo BB84 para 30 qubits, tanto con adversario como sin él. En el caso en que no hay interferencias, Alice y Bob forman sus claves eligiendo aquellos bits para los que han empleado las mismas bases. Como resultado, ambos consiguen una clave común de aproximadamente la mitad del número de qubits transmitidos.

Sin embargo, cuando Eve entra en juego, vemos que sus acciones modifican algunos bits de la clave compartida y es detectada. Por ejemplo, para el qubit correspondiente la segunda coincidencia de bases que tiene Alice y Bob, estos miden el qubit en la base Z y Eve lo mide en la base X. Esto causa que, aunque Alice haya mandado un 1 a Bob, Eve lo modifica a un estado de superposición. Por tanto, Bob tenía un 50 % de posibilidades de obtener 0 o 1, y en este caso ha obtenido un 0. Por tanto, la detección del espía por parte del protocolo se efectúa de manera adecuada, pues Eve había conseguido (teniendo un poco de suerte) más de la mitad de los bits que la conformaban.

En la figura 9.14 tenemos los siguientes resultados, correspondientes al protocolo B92 para 30 qubits. Tal y como esperamos, las claves resultantes de este protocolo son aproximadamente del tamaño del 25 % de los qubits transmitidos. En el caso sin espionaje de la figura este número se ve algo reducido, pero si ejecutamos varias veces el mismo vemos que es una simple anomalía estadística.

Cuando no hay espionaje, la clave se conforma por los qubits que Bob mide con valor 1, siendo un 0 si la medición se ha aplicado sobre el eje X y 1 si se ha aplicado sobre Z. Después, Bob publica los bits que conoce de la clave y Alice selecciona dichos bits de la transmisión que ha realizado. Como resultado, ambos tienen ahora la misma clave compartida.

Si Eve aplica la estrategia descrita, inevitablemente fallará reconstruyendo los qubits cuando mida como $|0\rangle$ el estado de superposición $|+\rangle$ y cuando mida como $|+\rangle$ el estado $|0\rangle$ en el eje X. Efectivamente, esto ocurre en el caso del último qubit que conforma la clave. En este, Alice envía un 1 codificado como $|+\rangle$. Eve lo mide en el eje Z y obtiene el estado $|0\rangle$. Siguiendo su esquema, manda dicho $|0\rangle$ a Bob. Este mide el estado en el eje X y obtiene el estado $|1\rangle$, por lo que deducirá correctamente que el estado que ha medido era originalmente $|0\rangle$. Cuando Alice y Bob comparen parte de sus claves, podrán detectar la acción de Eve viendo dicha diferencia.

Si ejecutamos el protocolo E91, también veremos que las salidas son las

```
-------Ejecucion sin espionaje-------
Ejes Alice:       [0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1]
Ejes Bob:         [1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1]
Coincidencias:    [0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1]
Mediciones Alice: [1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0]
Mediciones Bob:   [1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0]
Clave secreta Alice: [0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0]
Clave secreta Bob:   [0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0]
Las claves de Alice y Bob coinciden, por lo que no hay espionaje.

-------Ejecucion con espionaje-------
Ejes Alice:       [1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0]
Ejes Bob:         [0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1]
Coincidencias:    [0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 0]
Ejes Eve:         [1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1]
Mediciones Alice: [1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1]
Mediciones Bob:   [1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0]
Clave secreta Alice: [1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0]
Clave secreta Bob:   [1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0]
Clave secreta Eve:   [1, 2, 0, 2, 1, 1, 1, 2, 0, 0, 2, 1, 0]
Las claves de Alice y Bob NO coinciden: Eve ha sido detectada.
```

Figura 9.13: Resultados obtenidos en la ejecución de BB84 para 30 qubits. Arriba, resultados obtenidos sin espionaje. Abajo, resultados con espionaje activado. El estado 2 de un bit representa que es desconocido para el atacante.

deseadas si disponemos del número suficiente de qubits. En caso contrario, es posible que el análisis de los qubits que no se han medido en la misma base se vea afectado por la gran varianza existente por tener una muestra demasiado pequeña. De esta manera, se pueden dar tanto falsos positivos de la presencia del espía como falsos negativos donde Eve no es detectada.

Para analizar cuál es el tamaño mínimo del protocolo para que la detección de anomalías sea correcta, vamos a realizar el siguiente experimento: vamos a probar el circuito para diferente número de qubits transmitidos, y mediremos el porcentaje de coincidencias en las mediciones de Alice y Bob sobre los qubits que no han sido elegidos como clave. Analizaremos tanto el caso donde el protocolo se ejecuta correctamente como en el que Eve está presente.

Los resultados se ven reflejados en la figura 9.15. Vemos que, cuanto mayor es la muestra de qubits sobre la que se hace el cálculo del porcentaje de coincidencias, las medidas tienden a los valores teóricos del 25 % sin espionaje y del 37,5 % con espionaje. Sobre los 200 qubits, el protocolo presenta resultados suficientemente estables como para su correcto funcionamiento.

Con este ensayo de los protocolos, concluimos la parte experimantal del trabajo.

```
-------Ejecucion sin espionaje-------
Mensaje Alice:  [0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0]
Ejes Bob:       [1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1]
Mediciones Bob: [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
Aciertos:       [0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
Clave secreta Alice: [1, 0, 1, 0]
Clave secreta Bob:   [1, 0, 1, 0]
Las claves de Alice y Bob coinciden, por lo que no hay espionaje.

-------Ejecucion con espionaje-------
Mensaje Alice:  [0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1]
Ejes Bob:       [1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1]
Mediciones Bob: [0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
Aciertos:       [0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
Ejes Eve:       [0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1]
Mediciones Eve: [0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
Clave secreta Alice: [1, 1, 1, 0, 0, 0, 1]
Clave secreta Bob:   [1, 1, 1, 0, 0, 0, 0]
Clave secreta Eve:   [2, 2, 2, 2, 2, 0, 2]
Las claves de Alice y Bob NO coinciden: Eve ha sido detectada.
```

Figura 9.14: Resultados obtenidos en la ejecución de B92 para 30 qubits. Arriba, resultados obtenidos sin espionaje. Abajo, resultados con espionaje activado. El estado 2 de un bit representa que es desconocido para el atacante.

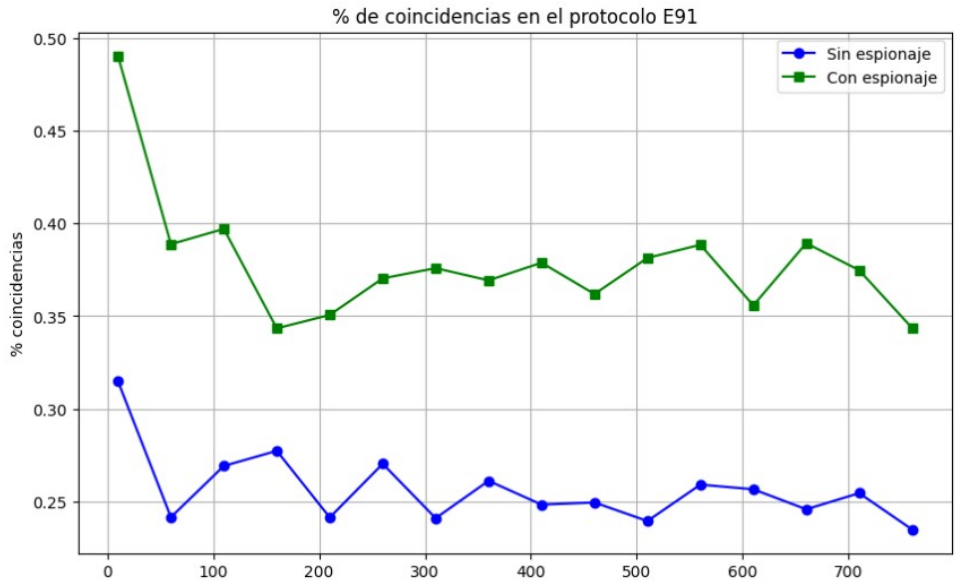

Figura 9.15: Porcentaje de coincidencias de las mediciones de los interlocutores en el protocolo E91 en función del número de qubits transmitidos.

Parte V

Conclusión

Capítulo 10

Conclusiones y trabajos futuros

10.1. Conclusión

En este Trabajo Fin de Grado, se ha realizado un profundo estudio sobre el algoritmo RSA y de su seguridad. Hemos tratado sobre sus detalles de implementación, como la elección de primos robustos o el uso de exponentes de cifrado y descifrado seguros, que le confieren la robustez que hace que sea uno de los criptosistemas más utilizados hoy en día. Además, hemos implementado diversos ataques clásicos sobre RSA como el método ρ de Pollard, la criba cuadrática y el ataque por fracciones continuas, cada uno de ellos basado en diferentes ideas y estructuras matemáticas.

Por otro lado, hemos explorado las posibilidades de la computación cuántica actual de comprometer la seguridad de RSA. Aunque los resultados en máquinas reales no hayan sido muy alentadores, la existencia del algoritmo de Shor sigue representando una grave amenaza al criptosistema. A través de simulaciones, hemos podido observar cómo este algoritmo puede hallar la factorización de un número en tiempo polinómico. Por tanto, si algún día la tecnología permite instalar un ordenador cuántico lo suficientemente potente como para ejecutarlo, la seguridad de RSA se vería prácticamente rota. Este hecho subraya la importancia del desarrollo de nuevos criptosistemas resistentes a los computadores cuánticos.

Por último, hemos estudiado la alternativa de los protocolos cuánticos de intercambio de clave y hemos desarrollado una prueba de concepto de ellos mediante simulaciones. BB84, B92 y E91, llegado el momento donde la tecnología lo permita, podrían proporcionar una solución segura a la compartición de claves en un mundo post-cuántico.

Personalmente, he disfrutado enormemente adentrándome en estos campos de conocimiento tan fascinantes. La criptografía y la computación cuántica son temáticas que, por lo general, pasan desapercibidas en nuestra carrera. Sin embargo, con este Trabajo Fin de Grado, he tenido la oportunidad de investigarlas a fondo y aportar mi granito de arena a la consolidación del conocimiento existente en estos ámbitos. Este proceso de aprendizaje, además de brindarme la satisfacción de adquirir nuevos saberes, me ha proporcionado una sólida base en áreas profesionales que actualmente no están en auge, pero que, en un futuro relativamente cercano, pueden ser altamente demandadas. Por lo tanto, este trabajo ha sido también una valiosa oportunidad para expandir mis horizontes y prepararme para los desafíos futuros en el campo de la criptografía y la computación cuántica.

10.2. Trabajos futuros

En el trascurso de este trabajo hemos dejado, por desgracia, muchas puertas de investigación interesantes sin abrir. Posibles continuaciones de este trabajo serían las siguientes:

1. Estudio e implementación del Algoritmo de Coppersmith y de la Criba General del Cuerpo de Números.

2. Estudio e implementación de otros protocolos de intercambio de claves cuánticos, como SARG04 o COW.

3. Estudio e implementación del "segundo paso" en algoritmos clásicos. Esto es, en algoritmos basados en producir un número K-potencia uniforme que divida a p, extender la implementación para proseguir con la ejecución del algoritmo si no se encuentra un factor tras aplicar la primera fase.

4. Implementación del método de factorización por curvas elípticas.

5. Optimización automática de parámetros para los algoritmos clásicos (p-1, curvas elípticas, criba cuadrática).

6. Implementar mejoras propuestas por [Mon+16] en el Algoritmo de Shor, donde se reduce el número de qubits y las puertas que conforman el circuito.

7. Explorar y desarrollar nuevos criptosistemas post-cuánticos resistentes a ataques clásicos y cuánticos.

Bibliografía

[Aca24] Khan Academy. *Fast Modular Exponentiation*. Accedido: 2024-04-09. 2024. URL: https://es.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/fast-modular-exponentiation.

[AKS04] Manindra Agrawal, Neeraj Kayal y Nitin Saxena. «PRIMES Is in P». En: *Annals of Mathematics* 160.2 (2004), págs. 781-793. ISSN: 0003486X. URL: http://www.jstor.org/stable/3597229 (visitado 29-05-2024).

[Bai19] Carlos Ruiz Baier. *Congruencias Módulo n*. Accessed: 2024-04-09. 2019. URL: https://blogs.mat.ucm.es/cruizb/wp-content/uploads/sites/48/2019/07/Z-Congruencias-7.pdf.

[BB14] Charles H. Bennett y Gilles Brassard. «Quantum cryptography: Public key distribution and coin tossing». En: *Theoretical Computer Science* 560 (dic. de 2014), págs. 7-11. ISSN: 0304-3975. DOI: 10.1016/j.tcs.2014.05.025. URL: http://dx.doi.org/10.1016/j.tcs.2014.05.025.

[Bea03] Stephane Beauregard. *Circuit for Shor's algorithm using 2n+3 qubits*. 2003. arXiv: quant-ph/0205095 [quant-ph].

[Ben+93] Charles H. Bennett et al. «Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels». En: *Phys. Rev. Lett.* 70 (13 mar. de 1993), págs. 1895-1899. DOI: 10.1103/PhysRevLett.70.1895. URL: https://link.aps.org/doi/10.1103/PhysRevLett.70.1895.

[Ben80] Paul Benioff. «The computer as a physical system: A microscopic quantum mechanical Hamiltonian model of computers as represented by Turing machines». En: *Journal of Statistical Physics* 22 (mayo de 1980), págs. 563-591. DOI: 10.1007/BF01011339.

[Ben92] Charles H. Bennett. «Quantum cryptography using any two nonorthogonal states». En: *Phys. Rev. Lett.* 68 (21 mayo de 1992), págs. 3121-3124. DOI: 10.1103/PhysRevLett.68.3121. URL:

https://link.aps.org/doi/10.1103/PhysRevLett.68.3121.

[Ber19] Chris Bernhardt. «Quantum Computing for Everyone». En: *Quantum Computing for Everyone*. 2019, págs. 1-12.

[BV97] Ethan Bernstein y Umesh Vazirani. «Quantum Complexity Theory». En: *SIAM Journal on Computing* 26.5 (1997), págs. 1411-1473. DOI: 10.1137/S0097539796300921. eprint: https://doi.org/10.1137/S0097539796300921. URL: https://doi.org/10.1137/S0097539796300921.

[BW92] Charles H. Bennett y Stephen J. Wiesner. «Communication via one- and two-particle operators on Einstein-Podolsky-Rosen states». En: *Phys. Rev. Lett.* 69 (20 nov. de 1992), págs. 2881-2884. DOI: 10.1103/PhysRevLett.69.2881. URL: https://link.aps.org/doi/10.1103/PhysRevLett.69.2881.

[DH76] W. Diffie y M. Hellman. «New directions in cryptography». En: *IEEE Transactions on Information Theory* 22.6 (1976), págs. 644-654. DOI: 10.1109/TIT.1976.1055638.

[DHM05] Raúl Durán, Luis Hernández y Jaime Muñoz. *El criptosistema RSA*. RA-MA, 2005. ISBN: 84-7897-651-5.

[DJ92] David Deutsch y Richard Jozsa. «Rapid Solution of Problems by Quantum Computation». En: *Proceedings of the Royal Society of London Series A* 439.1907 (dic. de 1992), págs. 553-558. DOI: 10.1098/rspa.1992.0167.

[DR98] Joan Daemen y Vincent Rijmen. «The Block Cipher Rijndael». En: vol. 1820. Ene. de 1998, págs. 277-284. ISBN: 978-3-540-67923-3. DOI: 10.1007/10721064_26.

[EFF98] EFF. *Cracking DES - Secrets of Encryption Research, Wiretap Politics & Chip Design*. Oreilly & Associates Inc., 1998. ISBN: 1-56592-520-3.

[EK03] Eli Biham Elad Barkan y Nathan Keller. «Instant Ciphertext-Only Cryptanalysis of GSM Encrypted Communication». En: *CRYPTO* (2003), págs. 600-616.

[Eke91] Artur K. Ekert. «Quantum cryptography based on Bell's theorem». En: *Phys. Rev. Lett.* 67 (6 ago. de 1991), págs. 661-663. DOI: 10.1103/PhysRevLett.67.661. URL: https://link.aps.org/doi/10.1103/PhysRevLett.67.661.

[Elg85] T. Elgamal. «A public key cryptosystem and a signature scheme based on discrete logarithms». En: *IEEE Transactions on Information Theory* 31.4 (1985), págs. 469-472. DOI: 10.1109/TIT.1985.1057074.

[Exc22] Stack Exchange. *The Ekert Protocol (E91) for Quantum Key Distribution*. https://quantumcomputing.stackexchange.com/questions/29068/the-ekert-protocol-e91-for-quantum-key-distribution. Último acceso: 31 de mayo de 2024. 2022.

[Fey82] Richard P Feynman. «Simulating physics with computers». En: *International journal of theoretical physics* 21.6/7 (1982), págs. 467-488.

[Gar18] Alicia Boya García. *Cifrado RC4*. 2018. URL: https://ntrrgc.me/attachments/Cifrado_RC4/.

[GNU24] GNU Project. *GNU MP: The GNU Multiple Precision Arithmetic Library Manual*. Último acceso: 21 de junio de 2024. 2024. URL: https://gmplib.org/manual/.

[Gor85] John Gordon. «Strong Primes are Easy to Find». En: *Advances in Cryptology*. Ed. por Thomas Beth, Norbert Cot e Ingemar Ingemarsson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1985, págs. 216-223. ISBN: 978-3-540-39757-1.

[HH00] L. Hales y S. Hallgren. «An improved quantum Fourier transform algorithm and applications». En: *Proceedings 41st Annual Symposium on Foundations of Computer Science*. 2000, págs. 515-525. DOI: 10.1109/SFCS.2000.892139.

[Ker83] A. Kerckhoffs. «La cryptographie militaire». En: *Journal des sciences militaires* (1883), págs. 5-38.

[Knu20] Donald Knuth. *Post on NMBRTHRY listserv*. Accessed: 2024-06-24. 2020. URL: https://listserv.nodak.edu/cgi-bin/wa.exe?A2=NMBRTHRY;dc42ccd1.2002.

[Lan61] R. Landauer. «Irreversibility and Heat Generation in the Computing Process». En: *IBM Journal of Research and Development* 5.3 (1961), págs. 183-191. DOI: 10.1147/rd.53.0183.

[Len87] H. W. Lenstra. «Factoring Integers with Elliptic Curves». En: *Annals of Mathematics* 126.3 (1987), págs. 649-673. ISSN: 0003486X. URL: http://www.jstor.org/stable/1971363 (visitado 05-05-2024).

[Mon+16] Thomas Monz et al. «Realization of a scalable Shor algorithm». En: *Science* 351.6277 (mar. de 2016), págs. 1068-1070. ISSN: 1095-9203. DOI: 10.1126/science.aad9480. URL: http://dx.doi.org/10.1126/science.aad9480.

[Mor03] Guillermo Morales-Luna. *Un poco de computación cuántica: Algoritmos más comunes*. 2003. URL: https://cs.cinvestav.mx/~gmorales/quantum/intro.pdf.

[NC10] M.A. Nielsen e I.L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010. ISBN: 9781139495486. URL: https://books.google.es/books?id=-s4DEy7o-a0C.

[PA22] Renato Portugal y Frank Acasiete. *Algoritmos Cuánticos Básicos*. Oct. de 2022.

[Pol74] J. M. Pollard. «Theorems on factorization and primality testing». En: *Mathematical Proceedings of the Cambridge Philosophical Society* 76.3 (1974), págs. 521-528. DOI: 10.1017/S0305004100049252.

[Pol75] John M. Pollard. «A monte carlo method for factorization». En: *BIT Numerical Mathematics* 15 (1975), págs. 331-334. URL: https://api.semanticscholar.org/CorpusID:122775546.

[Pom82] Carl Pomerance. «Analysis and comparison of some integer factoring algorithms». En: *Computational methods in Number Theory, Part I* (1982), págs. 89-131.

[Pom96] Carl Pomerance. «A tale of two sieves». En: *Notices of the American Mathematical Society* 43.12 (1996), págs. 1473-1485.

[Qis24] Qiskit Development Team. *Qiskit: An Open-source Quantum Computing Framework*. Último acceso: 31 de mayo de 2024. IBM Research. 2024. URL: https://docs.quantum.ibm.com.

[Qua23] IBM Quantum. *A New Quantum Metric: Layered Fidelity*. Último acceso: 18 de junio de 2024. 2023. URL: https://www.ibm.com/quantum/blog/quantum-metric-layer-fidelity.

[Qua24] IBM Quantum. *IBM Quantum Services and Resources*. Último acceso: 18 de junio de 2024. 2024. URL: https://quantum.ibm.com/services/resources.

[Rey18] Juan Arias de Reyna. *Los cuadrados y la factorización*. 2018. URL: https://institucional.us.es/blogimus/2018/10/los-cuadrados-y-la-factorizacion/.

[RSA78] Ron Rivest, Adi Shamir y Leonard Adleman. «A method for obtaining digital signatures and public-key cryptosystems». En: *Communications of the ACM* 21.2 (1978), págs. 120-126. URL: https://dl.acm.org/doi/pdf/10.1145/359340.359342.

[Sho99] Peter W. Shor. «Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer». En: *SIAM Review* 41.2 (1999), págs. 303-332. DOI: 10.1137/S0036144598347011. eprint: https://doi.org/10.1137/S0036144598347011. URL: https://doi.org/10.1137/S0036144598347011.

[SN98] National Institute of Standards y Technology (NIST). *Digital Signature Standard (DSS)*. Accessed: 2024-04-05. 1998. URL: https://web.archive.org/web/20131213131144/http://www.itl.nist.gov/fipspubs/fip186.htm.

[Ste09] William Stein. «Elementary Number Theory: Primes, Congruences, and Secrets: A Computational Approach». En: (ene. de 2009). DOI: 10.1007/b13279.

[VBE96] Vlatko Vedral, Adriano Barenco y Artur Ekert. «Quantum networks for elementary arithmetic operations». En: *Physical Review A* 54.1 (1996), pág. 147.

[Wac+21] Andrew Wack et al. *Quality, Speed, and Scale: three key attributes to measure the performance of near-term quantum computers*. 2021. arXiv: 2110.14108.

[Wei24] Eric W. Weisstein. *Prime Number Theorem*. Último acceso: 3 de mayo de 2024. 2024. URL: https://mathworld.wolfram.com/PrimeNumberTheorem.html.

[Wie90] M.J. Wiener. «Cryptanalysis of short RSA secret exponents». En: *IEEE Transactions on Information Theory* 36.3 (1990), págs. 553-558. DOI: 10.1109/18.54902.

[Wik24a] Wikipedia colaboradores. *Eliminación Gaussiana — Wikipedia, La Enciclopedia Libre*. Accedido: 2024-06-23. 2024. URL: https://en.wikipedia.org/wiki/Gaussian_elimination.

[Wik24b] Wikipedia contributors. *Tonelli–Shanks Algorithm — Wikipedia, The Free Encyclopedia*. Último acceso: 22 de junio de 2024. 2024. URL: https://en.wikipedia.org/wiki/Tonelli%E2%80%93Shanks_algorithm.

[Wil82] Hugh C. Williams. «A p+1 method of factoring». En: *Mathematics of Computation* 39 (1982), págs. 225-234. URL: https://api.semanticscholar.org/CorpusID:122364449.

[Won22] T.G. Wong. *Introduction to Classical and Quantum Computing*. Rooted Grove, 2022. ISBN: 9798985593105. URL: https://books.google.es/books?id=M3jqzgEACAAJ.