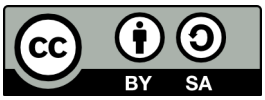
Universidad Rey Juan Carlos

CRIPTOGRAFÍA

Grado en Ingeniería de la Ciberseguridad

Libro de materiales (categorías 0, 1, 3, 6 y 7)
Guía de estudio · Apuntes de teoría ·
Colección de problemas resueltos · Banco
de preguntas · Software utilizado

Misael E. Marriaga



©2026 Misael E. Marriaga
Algunos derechos reservados
Esta obra está bajo una Licencia Creative Commons Atribución-CompartirIgual 4.0 Internacional
Disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

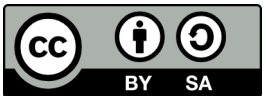
Autoría: Misael E. Marriaga

Material docente en abierto de la Universidad Rey Juan Carlos

Material docente en abierto de la Universidad Rey Juan Carlos

Asignatura:	Criptografía
Titulación:	Grado en Ingeniería de la Ciberseguridad
Código:	2285007
Curso:	2025-2026

Autoría:
Misael E. Marriaga
misael.marriaga@urjc.es



©2026 Misael E. Marriaga
Algunos derechos reservados
Este documento se distribuye bajo una licencia Creative Commons, Atribución-CompartirIgual 4.0 Internacional (CC BY-SA 4.0), disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Fecha de publicación: enero de 2026
Depósito: BURJC Digital https://burjcdigital.urjc.es (handle: se asigna al depositar)

Índice general

4 ····· **Parte I. Guía de estudio (Categoría 0)**
5 ····· Portada y datos rápidos
6 ····· Objetivos, temario oficial y carga de trabajo
7 ····· Evaluación ordinaria y extraordinaria
8 ····· Plan de estudio (calendario 2025-26)
9 ····· Consejos, normas, bibliografía y checklist
10 ····· **Parte II. Apuntes de teoría (Categoría 1)**
11 ····· Portada y ficha de licencia/depósito
13 ····· Tabla de contenidos (apuntes)
21 ····· Introducción (apuntes)
25 ····· Capítulo 1. Notación y repaso de conceptos
33 ····· Capítulo 2. Algoritmos y complejidad
63 ····· Capítulo 3. Introducción al pensamiento matemático
87 ····· Capítulo 4. El One-time pad y el Principio de Kerckhoffs
97 ····· Capítulo 5. Fundamentos de seguridad demostrable
121 ····· Capítulo 6. Basando la criptografía en cálculos intratables
137 ····· Capítulo 7. Generadores pseudoaleatorios
151 ····· Capítulo 8. Funciones pseudoaleatorias y cifrados en bloque
169 ····· Capítulo 9. Seguridad frente a ataques de texto plano elegido
179 ····· Capítulo 10. Modos de operación de cifrados en bloque
189 ····· Capítulo 11. Texto cifrado escogido
205 ····· Capítulo 12. Códigos de autenticación de mensajes
217 ····· Capítulo 13. Funciones Hash
223 ····· Capítulo 14. RSA y firmas digitales
237 ····· Capítulo 15. Establecimiento de clave de Diffie-Hellman
243 ····· Capítulo 16. Cifrado de clave pública
253 ····· Capítulo 17. Compartición de secretos
271 ····· Capítulo 18. Esquemas de compromiso
287 ····· **Parte III. Ejercicios resueltos (Categoría 3)**
288 ····· Portada y ficha de licencia/depósito
290 ····· Hoja de problemas 1: Intercambiabilidad de librerías y distinguidoras
297 ····· Hoja de problemas 2: OTP y PRG: distinguidoras y cierres
306 ····· Hoja de problemas 3: Modos de operación y seguridad CPA/CCA
316 ····· Hoja de problemas 4: MAC y propiedades de funciones hash
325 ····· Hoja de problemas 5: Establecimiento de clave, DDH/ElGamal y firmas RSA
331 ····· Hoja de problemas 6: Compartición de secretos (Shamir), umbrales y seguridad
339 ····· Hoja de problemas 7: Compromisos: hash, PRG y RSA; reducciones
346 ····· **Parte IV. Banco de Preguntas para Moodle (Categoría 7)**
347 ····· Portada y ficha de licencia/depósito
349 ····· Banco de preguntas (GIFT)
349 ····· Bloque 1: Fundamentos y otp (P01–P03)
349 ····· Bloque 2: Simétrica (prg, prf, modos) (P04–P07)
350 ····· Bloque 3: Integridad y hash (P08–P11)
350 ····· Bloque 4: Asimétrica y teoría de números (P12–P15)
350 ····· Bloque 5: Firmas y conceptos avanzados (P16–P20)
352 ····· **Parte V. Software utilizado en la asignatura (Categoría 6)**
353 ····· Ficha del software

PARTE I

Guía de estudio (Categoría 0)

GUÍA DE ESTUDIO

CRIPTOGRAFÍA · Grado en Ingeniería de la Ciberseguridad (Móstoles)

Curso académico 2025-2026 · 2º cuatrimestre · 6 ECTS · Asignatura obligatoria

Información rápida

Código	2285007
Grupo	Mañana A (1º curso)
Horario	Lunes 13:00–15:00 · Miércoles 11:00–13:00
Aula	Laboratorios Polivalentes II – Aula de Informática 007
Profesorado	Misael E. Marriaga
Contacto y tutorías	Por correo electrónico (se acuerda horario)

Correos:

- misael.marriaga@urjc.es

©2026 Misael E. Marriaga
Algunos derechos reservados
Esta obra está bajo una Licencia Creative Commons Atribución-CompartirIgual 4.0 Internacional
Disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Autoría: Misael E. Marriaga

Material docente en abierto de la Universidad Rey Juan Carlos

1. Cómo usar esta guía

Esta guía resume qué se evalúa, cómo se organiza el curso y qué rutina de estudio suele funcionar mejor. La planificación semanal es orientativa y puede ajustarse por festivos o necesidades docentes. El Aula Virtual es el canal oficial de avisos y cambios.

2. Objetivos de aprendizaje

Al finalizar la asignatura deberías ser capaz de:

- Explicar los objetivos clásicos de la criptografía (confidencialidad, integridad, autenticación) y su relación con modelos de amenaza.
- Interpretar definiciones de seguridad modernas y saber qué significa ‘seguridad demostrable’ en términos prácticos.
- Aplicar herramientas de criptografía simétrica (cifrado, hash, MAC) y reconocer usos inseguros frecuentes.
- Comprender los mecanismos básicos de criptografía asimétrica (cifrado, intercambio de claves, firmas) y su papel en sistemas reales.
- Resolver ejercicios teórico-prácticos con precisión: algoritmos, condiciones, ataques y justificación del nivel de seguridad.

3. Temario oficial (mapa del curso)

Bloque I

- Tema 1. (1.1) Introducción a la criptografía; (1.2) Fundamentos de seguridad demostrable.
- Tema 2. (2.1) Criptografía clásica; (2.2) Cifrado simétrico; (2.3) Funciones hash y MAC.

Bloque II

- Tema 3. (3.1) Cifrado asimétrico; (3.2) Intercambio de clave; (3.3) Firma digital.
- Tema 4. (4.1) Esquemas de compromiso; (4.2) Compartición de secretos.

4. Tiempo de trabajo y rutina recomendada

Carga total prevista: 180 horas de trabajo del estudiante (6 ECTS). Distribución orientativa:

Clases teóricas	25 h
Clases de resolución de ejercicios	20 h
Prácticas (laboratorio/tecnológicas)	10 h
Realización de pruebas	5 h
Tutorías académicas	14 h
Actividades relacionadas (jornadas, seminarios)	4 h
Preparación de teoría	36 h
Preparación de prácticas/ejercicios	36 h
Preparación de pruebas	30 h
Total	**180 h**

Material docente en abierto de la Universidad Rey Juan Carlos

Rutina semanal mínima (recomendada)

Para llegar bien a las pruebas, la constancia semanal importa más que 'maratones' al final:

- Antes de clase (30–45 min): lectura rápida del tema y 3 dudas concretas.
- Después de clase (60–90 min): resumen de 1 página + 3–5 ejercicios cortos.
- Cada 2 semanas: un repaso activo (explicar sin mirar apuntes, detectar huecos).

5. Evaluación (ordinaria)

El modelo general es evaluación continua. Todas las pruebas son reevaluables en convocatoria extraordinaria. Para aprobar en ordinaria deben cumplirse mínimos por bloques y mínimo global.

Actividades y ponderación

Actividad	Bloque	Peso	Formato	Duración aprox.	Cuándo
T1 (Test 1)	I	10% (0–1)	Online (Aula Virtual)	45 min	Semana ~10
P1 (Parcial 1)	I	40% (0–4)	Presencial	90 min	Semana ~10
T2 (Test 2)	II	10% (0–1)	Online (Aula Virtual)	45 min	Última semana con clases (~15)
P2 (Parcial 2)	II	40% (0–4)	Presencial	90 min	Periodo oficial de exámenes (ordinaria)

Mínimos para aprobar en ordinaria

- P1 + T1 ≥ 2 puntos sobre 5 (Bloque I).
- P2 + T2 ≥ 2 puntos sobre 5 (Bloque II).
- P1 + P2 + T1 + T2 ≥ 5 puntos sobre 10 (nota final).

6. Convocatoria extraordinaria (si no se supera en ordinaria)

En extraordinaria solo se reevalúan las actividades reevaluables no superadas. Si se han superado los mínimos, pero la nota global no alcanza 5, de forma excepcional se puede revaluar alguna actividad reevaluable con nota < 5.

Pruebas y sustitución de notas

- F1 (Extraordinaria Bloque I): teoría y ejercicios del Bloque I. Peso 50%. Obligatoria si P1 + T1 < 2; optativa en otro caso. La nota (0–5) sustituye a P1 + T1.
- F2 (Extraordinaria Bloque II): teoría y ejercicios del Bloque II. Peso 50%. Obligatoria si P2 + T2 < 2; optativa en otro caso. La nota (0–5) sustituye a P2 + T2.
- Las pruebas se realizan durante el periodo oficial de exámenes de extraordinaria. Duración aproximada: 90 min cada una.

Mínimos para aprobar en extraordinaria

- F1 ≥ 2 puntos sobre 5.
- F2 ≥ 2 puntos sobre 5.
- F1 + F2 ≥ 5 puntos sobre 10.

7. Plan de estudio (calendario 2025–26)

La siguiente planificación semanal incluye fechas, temas y prácticas. Las referencias a "Capítulo" corresponden al material base del curso. Salvo aviso en Aula Virtual, las sesiones se distribuyen en lunes (13:00–15:00) y miércoles (11:00–13:00).

Semana (fechas)	Lunes (13:00–15:00)	Miércoles (11:00–13:00)
Semana 1: 19 al 23 de enero	Periodo de evaluación	Periodo de evaluación
Semana 2: 26 al 30 de enero	Presentación (Introducción)	FESTIVO
Semana 3: 2 al 6 de febrero	One-time Pad y Principio de Kerckhoffs (Capítulo 4)	Fundamentos de seguridad demostrable (Capítulo 5,6)
Semana 4: 9 al 13 de febrero	Generadores pseudoaleatorios (Capítulo 7)	Funciones pseudoaleatorias y cifrados en bloques (Capítulo 8)
Semana 5: 16 al 20 de febrero	Seguridad CPA (Capítulo 9)	Práctica 1 – MAGISTRAL
Semana 6: 23 al 27 de febrero	Modos de operación para cifrados en bloque (Capítulo 10)	Práctica 2 – MAGISTRAL
Semana 7: 2 al 6 de marzo	Seguridad CCA (Capítulo 11)	Práctica 3 – AUTODIRIGIDA
Semana 8: 9 al 13 de marzo	Códigos MAC (Capítulo 12)	Práctica 4 – MAGISTRAL
Semana 9: 16 al 20 de marzo	Funciones Hash (Capítulo 13)	Práctica 5 – MAGISTRAL
Semana 10: 23 al 27 de marzo	Práctica 6- AUTODIRIGIDA	PRIMER PARCIAL
VACACIONES DE SEMANA SANTA		
Semana 11: 6 al 10 de abril	FESTIVO	RSA y firmas digitales (Capítulo 14)
Semana 12: 13 al 17 de abril	Establecimiento de claves (Capítulo 15)	Práctica 7 – MAGISTRAL
Semana 13: 20 al 24 de abril	Cifrado de clave pública (Capítulo 16)	Práctica 8 – AUTODIRIGIDA
Semana 14: 27 de abril a 1 de mayo	Compartición de secretos (Capítulo 17)	Práctica 9 – MAGISTRAL
Semana 15: 4 a 8 de mayo	Esquemas de compromiso (Capítulo 18)	Práctica 10 – AUTODIRIGIDA

Nota: la fecha exacta del PRIMER PARCIAL y cualquier ajuste por festivos se publicará en el Aula Virtual.

8. Consejos de estudio que suelen marcar la diferencia

En criptografía se aprueba 'pensando con precisión'. Recomendaciones prácticas:

- Estudia por capas: (1) qué se quiere garantizar, (2) modelo de ataque, (3) construcción, (4) por qué resiste (o cómo se rompe).
- Después de cada ejercicio, pregúntate: '¿qué condición es crítica?' y '¿qué ataque aparece si la quito?'
- Entrena escritura: define variables, escribe claramente entradas/salidas y no dejes pasos 'implícitos'.
- No acumules semanas: si una definición no está clara, lo demás se vuelve cuesta arriba.

9. Normas y situaciones especiales (resumen)

Dispensa Académica de Asistencia a Clase (DAAC)

La DAAC no exime automáticamente de la evaluación continua ni de actividades presenciales obligatorias. Si te conceden DAAC, debes contactar con el profesorado para acordar adaptaciones que garanticen los resultados de aprendizaje y su evaluación.

Integridad académica

La Universidad mantiene estándares altos de integridad y honestidad académica y puede utilizar herramientas de supervisión y antiplagio. Trabaja con transparencia y cita fuentes si reutilizas material.

10. Bibliografía recomendada

Bibliografía básica:

- Nigel Smart, Cryptography: an Introduction (disponible gratuitamente online).
- M.I. González Vasco y A. Pérez del Pozo, Criptografía Esencial (RAMA, 2021).
- J. Katz y Y. Lindell, Introduction to Modern Cryptography (2014).

Bibliografía complementaria:

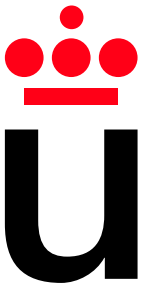
- S. Vaudenay, A Classical Introduction to Cryptography (Springer, 2006).
- N.P. Smart, Cryptography Made Simple (Springer, 2016).

11. Checklist rápida antes de cada prueba

- Puedo definir (sin mirar apuntes) las nociones clave del bloque correspondiente.
- He resuelto ejercicios variados: cortos (concepto) y largos (procedimiento).
- Sé escribir algoritmos con entradas/salidas y justificar por qué se cumple la propiedad.
- He revisado errores típicos (claves repetidos, uso indebido de hash/MAC, supuestos no mencionados, etc.).

PARTE II

Apuntes de teoría (Categoría 1)



Universidad
Rey Juan Carlos

CRIPTOGRAFÍA

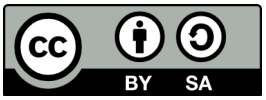
Grado en Ingeniería de la Ciberseguridad

Misael E. Marriaga

Material docente en abierto de la Universidad Rey Juan Carlos

Asignatura:	Criptografía
Titulación:	Grado en Ingeniería de la Ciberseguridad
Código:	2285007
Curso:	2025-2026

Autoría:
Misael E. Marriaga
misael.marriaga@urjc.es



©2026 Misael E. Marriaga
Algunos derechos reservados
Este documento se distribuye bajo una licencia Creative Commons, Atribución-CompartirIgual 4.0 Internacional (CC BY-SA 4.0), disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Tabla de contenidos

Introducción **xi**

1. Notación y repaso de conceptos **1**

1.1. Logaritmos y exponentes 1

1.2. Aritmética modular 1

1.3. Cadenas de bits 5

1.4. Funciones 6

1.5. Probabilidad 6

1.5.1. Consejos y trucos 6

1.5.2. Terminología precisa 7

2. Algoritmos y complejidad **9**

2.1. Algoritmos 10

2.2. Notación de pseudocódigo 11

2.3. Algoritmos de búsqueda 12

2.3.1. Búsqueda lineal 12

2.3.2. Búsqueda binaria 13

2.4. Algoritmos de ordenación 15

2.4.1. Método de la burbuja 15

2.4.2. Ordenación por inserción 16

2.5. Algoritmos voraces 17

2.6. Crecimiento de funciones 20

2.6.1. La notación O mayúscula 21

2.6.2. Algunos resultados importantes sobre la notación O 24

2.6.3. Crecimiento de combiandción de funciones 26
2.6.4. Las notaciones Omega y Zeta 28
2.7. Complejidad de algoritmos 31
2.7.1. Complejidad en tiempo 31
2.7.2. Complejidad del peor caso 32
2.7.3. Complejidad del caso promedio 33
2.7.4. Complejidad del peor caso de dos algoritmos de ordenación 34
2.7.5. El significado de la complejidad de un algoritmo 35

3. Introducción al pensamiento matemático **39**
Definiciones 42
Teoremas 44
Demostraciones 51
Algunos consejos 60

4. El One-time pad y el Principio de Kerckhoffs **63**
4.1. ¿Qué es (y qué no es) la criptografía? 63
4.1.1. Conceptos básicos y terminología de cifrado 63
4.1.2. Secretos y el Principio de Kerckhoffs 64
4.1.3. Excusas, excusas 65
4.1.4. No criptografía 66
4.2. Especificidades del One-time pad 66
4.2.1. Corrección 67
4.2.2. Seguridad 68
4.2.3. Discusión 70
4.2.4. Limitaciones 71

5. Fundamentos de seguridad demostrable **73**
5.1. Cómo escribir una definición de seguridad 73
5.1.1. Lo que no forma parte de una definición de seguridad 74
5.2. Definición de seguridad al estilo “Real-vs-Aleatorio” 75

5.2.1. Definición de seguridad al estilo "Intercambiabilidad" 76
5.3. Formalismos para Definiciones de Seguridad 77
5.3.1. Librerías intercambiables 79
5.4. Reformulaciones formales de conceptos anteriores 80
5.4.1. Discusión, Errores comunes 82
5.4.2. El Principio de Kerckhoffs, Revisado 83
5.5. Cómo demostrar inseguridad con ataques 84
5.6. Cómo demostrar seguridad 87
5.6.1. Encadenando varios componentes 87
5.6.2. Un ejemplo de demostración 88
5.6.3. Resumen 90
5.6.4. Un Ejemplo contrastante 91
5.7. Cómo comparar/contrastar definiciones de seguridad 92
5.7.1. Una definición de seguridad implica otra 92
5.7.2. Una definición de seguridad no implica otra 94

6. Basando la criptografía en cálculos intratables **97**
6.1. ¿Qué califica como un ataque "computacionalmente inviable"? 98
6.1.1. Tiempo asintótico de ejecución 99
6.1.2. Trampa potencial: algoritmos numéricos 99
6.2. ¿Qué califica como una probabilidad de éxito "despreciable"? 100
6.3. Indistinguibilidad 102
6.3.1. Otras propiedades 104
6.3.2. Lema del suceso malo 104
6.4. Probabilidades de cumpleaños y muestreo con/sin reemplazo 106
6.4.1. Probabilidades de cumpleaños 106
6.4.2. Acotaciones asintóticas en la probabilidad del cumpleaños 108
6.4.3. El problema del cumpleaños en términos de librerías indistinguibles 110

7. Generadores pseudoaleatorios **113**
7.1. Definiciones 114

7.2. Generadores Pseudoaleatorios en la práctica . . . 116
7.2.1. Cómo NO construir un PRG . . . 116
7.2.2. Concepto relacionado: generación de números aleatorios . . . 118
7.3. Aplicación: claves más cortas en el one-time pad . . . 118
7.4. Extendiendo la expansión de un PRG . . . 121
7.4.1. Dos enfoques para aumentar el estiramiento . . . 121
7.4.2. Una prueba de seguridad . . . 121
7.4.3. Dónde falla la prueba para H_2 . . . 124
7.4.4. Un ataque concreto contra H_2 . . . 125

8. Funciones pseudoaleatorias y cifrados en bloque 127
8.1. Definición . . . 128
8.1.1. Cómo NO construir una PRF . . . 130
8.2. PRFs vs PRGs . . . 132
8.2.1. Construyendo un PRG a partir de una PRF . . . 132
8.3. Cifrados en bloque (permutaciones pseudoaleatoria) . . . 137
8.4. Relacionando PRFs y cifrados en bloque . . . 138
8.4.1. Lema de conmutación (¡Las PRPs también son PRFs!) . . . 138
8.5. Construyendo una PRP a partir de una PRF: La construcción de Feistel . . . 140
8.6. PRFs y cifrados en bloque en la práctica . . . 143

9. Seguridad frente a ataques de texto plano elegido 145
9.1. Límites del cifrado determinista . . . 146
9.1.1. Imposibilidad del cifrado determinista . . . 146
9.1.2. Evitando el cifrado determinista . . . 147
9.2. Textos cifrados pseudoaleatorios . . . 148
9.3. Cifrado seguro bajo CPA basado en PRFs . . . 150

10. Modos de operación de cifrados en bloque 155
10.1. Un recorrido por los modos comunes . . . 155
10.1.1. ECB: Electronic codebook (¡NUNCA, NUNCA USES ESTO! ¡NUNCA!) . . . 155

10.1.2. CBC: Encadenamiento de bloques cifrados . . . 156
10.1.3. CTR: Contador . . . 157
10.1.4. OFB: Retroalimentación de salida . . . 158
10.1.5. Comparar y contrastar . . . 158
10.2. Seguridad CPA y textos planos de longitud variable . . . 159
10.2.1. ¡No des por sentada la fuga de longitud! . . . 161
10.3. Seguridad del modo OFB . . . 162

11. Texto cifrado escogido **165**
11.1. Ataques de Oráculo de Padding . . . 165
11.1.1. Maleabilidad del Cifrado CBC . . . 166
11.1.2. Aprendiendo el último byte de un bloque . . . 167
11.1.3. Aprendiendo otros bytes de un bloque . . . 168
11.2. ¿Qué salió mal? . . . 169
11.2.1. Maleabilidad CPA . . . 172
11.3. Definición de seguridad CCA . . . 175
11.3.1. Un ejemplo . . . 176
11.3.2. Discusión . . . 177
11.3.3. Textos cifrados pseudoaleatorios . . . 179

12. Códigos de autenticación de mensajes **181**
12.1. Definición . . . 181
12.1.1. Cómo pensar sobre las propiedades de autenticidad . . . 182
12.1.2. La definición de seguridad del MAC . . . 182
12.1.3. Aplicaciones MAC . . . 183
12.2. MACs para mensajes largos . . . 185
12.2.1. Cómo NO hacerlo . . . 185
12.2.2. Cómo hacerlo: CBC-MAC . . . 186
12.2.3. CBC-MAC más bobusto . . . 187
12.3. Cifrar-luego-MAC . . . 188

13. Funciones Hash **193**

13.1. Propiedades de seguridad para funciones hash . . . 193

13.1.1. Ataques de fuerza bruta a funciones hash . . . 194

13.1.2. Seguridad de la función hash en la práctica . . . 195

13.1.3. Seguridad de la función hash en teoría . . . 196

13.1.4. Sal en la práctica . . . 197

14. RSA y firmas digitales **199**

14.1. División módulo n . . . 199

14.1.1. Inversos Multiplicativos . . . 200

14.1.2. La función ϕ de Euler . . . 202

14.2. La función RSA . . . 203

14.2.1. Cómo [No] exponenciar números enormes . . . 203

14.2.2. Propiedades de seguridad y discusión . . . 205

14.3. Firmas digitales . . . 206

14.3.1. Firmas RSA de "libro de texto" . . . 207

14.3.2. Firmas RSA con hash . . . 208

14.4. La dificultad de factorizar N . . . 209

14.4.1. Usando raíces cuadradas de la unidad para factorizar N . . . 209

14.4.2. Encontrando raíces cuadradas de la unidad . . . 210

15. Establecimiento de clave de Diffie-Hellman **213**

15.1. Grupos cíclicos . . . 213

15.1.1. Logaritmos discretos . . . 214

15.2. Establecimiento de claves de Diffie-Hellman . . . 214

15.2.1. Definiendo la seguridad para el establecimiento de claves . . . 215

15.3. Problema de Diffie-Hellman decisional . . . 216

15.3.1. ¿Para qué grupos se cumple la suposición DDH? . . . 216

16. Cifrado de clave pública **219**

16.1. Definiciones de seguridad . . . 219

16.1.1. Textos cifrados pseudoaleatorios . . . 220
16.2. La seguridad de un solo uso implica la seguridad de múltiples usos . . . 221
16.3. Cifrado ElGamal . . . 223
16.3.1. Seguridad . . . 224
16.4. Seguridad CCA y maleabilidad . . . 226

17. Compartición de secretos **229**
17.1. Definiciones . . . 230
17.2. Definición de seguridad . . . 230
17.2.1. Un enfoque inseguro . . . 232
17.3. Un esquema simple 2 de 2 . . . 233
17.4. Interpolación polinómica . . . 236
17.4.1. Polinomios sobre los Rreales . . . 237
17.4.2. Polinomios mod p . . . 239
17.5. Compartición de secretos de Shamir . . . 241
17.5.1. Seguridad . . . 242

18. Esquemas de compromiso **247**
18.1. Introducción . . . 247
18.2. Definición Formal y Seguridad . . . 248
18.3. Esquemas de compromiso basado en una función hash . . . 250

Librerías **255**

Introducción

Este manuscrito contiene los apuntes de la asignatura Criptografía del Grado en Ingeniería de la Ciberseguridad.

¿De qué trata este curso?

Este curso trata sobre los fundamentos de la seguridad demostrable.

- **Seguridad:** La criptografía se ocupa del control del acceso a la información. Desglosamos el concepto amplio de "seguridad" en objetivos más específicos: confidencialidad, autenticidad, e integridad.
- **Demostrable:** Podemos definir formalmente qué significa ser seguro y, luego, probar matemáticamente afirmaciones sobre la seguridad. Un tema destacado en el curso es la lógica de componer bloques de construcción de manera segura.
- **Fundamentos:** Este es un curso introductorio sobre el tema que cubre los conceptos básicos. Al completar este curso, contarás con una sólida base teórica que podrás aplicar en la mayoría de situaciones reales. También estarás preparado para estudiar temas más avanzados en criptografía.

Este manuscrito no es un manual que indique qué algoritmo criptográfico utilizar en cada situación, ni una guía para implementar de forma segura librerías criptográficas prefabricadas. No se discuten programas criptográficos específicos (por ejemplo, PGP, Tor, Signal, TrueCrypt) ni criptomonedas como Bitcoin. Tampoco aprenderás a convertirte en hacker leyendo este manuscrito.

¿A quién va dirigido este curso?

Este curso es para cualquier persona que necesite asegurar información mediante criptografía y tenga curiosidad por entender qué hace que algunas cosas sean "seguras" (y otras inseguras). Es poco probable que la mayoría de los lectores de este manuscrito desarrollen su propia criptografía

novedosa (por ejemplo, diseñando nuevos cifrados en bloque), pero es mucho más probable que utilicen y combinen bloques criptográficos, de ahí nuestro enfoque en la lógica de la composición.

¿Qué conocimientos previos se necesitan para entender este curso?

Sacarás el máximo provecho de este curso si tienes una base sólida en los contenidos estándar impartidos en las siguientes asignaturas:

- **Matemáticas discretas:** El curso requiere que estés familiarizado con aritmética modular básica, probabilidades discretas, y combinatoria simple.
- **Lógica:** La lógica de proposiciones y de predicado de primer orden son fundamentales para poder demostrar de manera rigurosa afirmaciones sobre seguridad a partir de premisas y definiciones formales.
- Se recomienda encarecidamente tener conocimientos de **algoritmos y estructuras de datos**, así como de **teoría de la computación** (autómatas, lenguajes formales y computabilidad). Trabajamos con cálculos y algoritmos en un alto nivel de abstracción y con rigor matemático, por lo que la experiencia previa con este estilo de pensamiento será útil.

¿Por qué la criptografía es un tema difícil?

¿Es por las matemáticas, verdad? La criptografía tiene fama de ser un tema complicado debido a la cantidad de matemáticas difíciles, pero creo que esta evaluación no es del todo acertada. Un antiguo estudiante lo resumió de manera clara cuando compartió en clase (parafraseado):

Algunos estudiantes estaban pensando en tomar tu curso, pero les preocupaba que incluyera demasiadas matemáticas. Yo no diría que este curso es exactamente muchas matemáticas. ¡De alguna manera es aún peor!

Gracias, creo.

En fin, muchos aspectos de la criptografía emplean matemáticas que a la mayoría de los estudiantes del Grado en Ingeniería de la Ciberseguridad les parecerían bastante avanzadas (algoritmos avanzados de factorización, curvas elípticas, isogenias, incluso geometría algebraica), pero estos no son el enfoque de este curso. Nuestro objetivo es, en cambio, la lógica de componer distintos bloques de construcción de manera demostrablemente segura. Sí, probablemente aprenderás algunas matemáticas nuevas en este curso —lo suficiente para entender RSA, por ejemplo—. Y sí, habrá bastantes "demostraciones". Pero, honestamente, creo que te irá bien si te fue bien en un curso estándar de matemáticas discretas. A mis estudiantes de Criptografía os digo que no espero que amen las matemáticas, las demostraciones y la teoría, solo os pido que decidan no temerles.

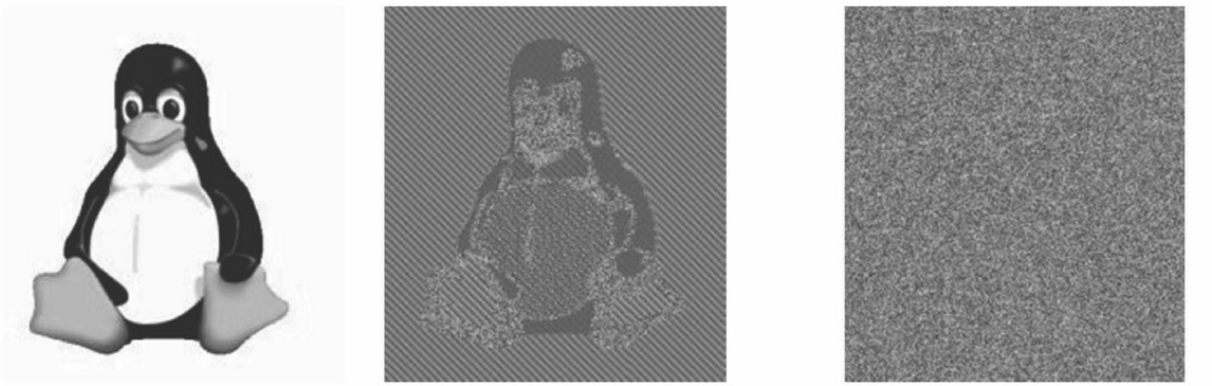


Figura 1: Ejemplo de un sistema inseguro. La figura central muestra cómo puede fallar un sistema inseguro al intentar ocultar el contenido de la imagen de la izquierda. La figura de la derecha es la salida de un sistema seguro que intenta ocultar la misma imagen.

¿Si no son las matemáticas, entonces qué? En un curso de algoritmos, puedo introducir y explicar conceptos con ejemplos concretos – esto es lo que ocurre paso a paso al ejecutar un algoritmo de ordenación en un array específico, esto es lo que ocurre al ejecutar el algoritmo de Dijkstra en un grafo en particular, estos son 42 ejemplos de un árbol de expansión. Puedes estudiar estos ejemplos concretos o incluso crear los tuyos propios para desarrollar una comprensión del caso general. La criptografía es diferente porque nuestras principales preocupaciones están en un **nivel de abstracción superior** al que la mayoría de los estudiantes suelen estar cómodos.

Sí, puedo ilustrar lo que ocurre paso a paso cuando ejecutas un algoritmo criptográfico en una entrada específica. Esto podría ayudarte a entender **qué hace el algoritmo**, pero nunca podrá ilustrar **por qué el algoritmo es seguro**. Esta pregunta del "por qué" es el enfoque principal de este curso.

- **Seguridad** es una propiedad global sobre el comportamiento de un sistema en todos los posibles valores de entrada. No puedes demostrar la seguridad con ejemplos, y no es suficiente ver qué ocurre en una ejecución particular de un algoritmo. La seguridad se trata de un nivel de abstracción superior.
- La mayoría de las definiciones de seguridad en este curso son esencialmente: *el sistema es seguro si sus salidas parecen datos aleatorios*. Si doy un ejemplo que sea lo suficientemente concreto para mostrar entradas y salidas reales, y si todo funciona correctamente, entonces todas las salidas solo parecerán basura sin sentido. Lamentablemente, nadie ha aprendido mucho mirando basura sin sentido.

Los sistemas son *inseguros* cuando sus salidas fallan en parecerse adecuadamente a datos alea-

torios. Ocasionalmente, fallan tan espectacularmente que puedes notarlo al observar los valores concretos de entrada y salida (ver la Figura 1). Pero, más a menudo, la razón de la inseguridad no es evidente. Por ejemplo, supón que un esquema de cifrado es inseguro porque el xor de los dos primeros bloques de salida es el mismo que el xor del tercer y cuarto bloque de salida. No creo que sea útil mostrar valores concretos con esta propiedad. Además, en algunos casos, la razón de la inseguridad solo "salta a la vista" en elecciones específicas y poco obvias de entradas.

Si quieres estar preparado para responder preguntas como "¿por qué esta cosa es segura, pero otra cosa muy similar no lo es?", entonces debes desarrollar una comprensión en este nivel superior de abstracción. Deberás enfrentar directamente ideas abstractas como "las salidas de este algoritmo parecen datos aleatorios, en estas circunstancias" y las implicaciones de estas ideas. Es difícil llegar a una comprensión sin el apoyo usual de ejemplos concretos (ver algoritmos ejecutados en entradas específicas), pero este manuscrito es mi mejor esfuerzo por hacer el camino lo más claro posible.

Capítulo 1

Notación y repaso de conceptos

El contenido de este capítulo es un repaso de conceptos que será útil para el resto del curso.

1.1. Logaritmos y exponentes

Probablemente aprendiste (y luego olvidaste) estas identidades en la secundaria o bachillerato:

- $(x^a)\,(x^b) = x^{a+b}$
- $(x^a)^b = x^{ab}$
- $\log_x(ab) = \log_x a + \log_x b$
- $a \log_x b = \log_x(b^a)$

Bueno, es momento de volver a familiarizarse con ellas.

En particular, **nunca** escribas $(x^a)\,(x^b) = x^{ab}$. Si escribes esto, tu profesor de Criptografía se dará cuenta de que la vida es demasiado corta, renunciará de inmediato a la enseñanza y se unirá a un circo ambulante. Pero no antes de cambiar tu calificación en el curso a un cero.

1.2. Aritmética modular

Escribiremos el conjunto de número enteros como:

$$\mathbb{Z} := \{\ldots, -2, -1, 0, 1, 2, \ldots\},$$

y el conjunto de los número naturales (enteros no negativos) como:

$$\mathbb{N} := \{0, 1, 2, \ldots\}.$$

En este curso, el 0 es considerado un número natural.

Definición 1.2.1. *Para $x, n \in \mathbb{Z}$, diremos que n **divide** a x (o que x es un **múltiplo** de n), y escribiremos $n|x$, si existe un entero k tal que $x = k\,n$.*

Recuerda que las definiciones se aplican tanto a números positivos como negativos (y al cero). En general, solo nos importa esta definición en el caso en que n es positivo, pero es común considerar tanto valores positivos como negativos de x.

Ejemplo 1.1.
7 divide a 84 porque podemos escribir $84 = 12 \cdot 7$.

7 divide a 0 porque podemos escribir $0 = 0 \cdot 7$.

7 divide a -77 porque podemos escribir $-77 = (-11) \cdot 7$.

-7 divide a 42 porque podemos escribir $42 = (-6) \cdot (-7)$.

1 divide a cada entero; -1 también divide a cada entero.

El único entero que divide a 0 es él mismo.

Definición 1.2.2. *Sean $n \in \mathbb{N}$ y $a \in \mathbb{Z}$ con $n \geq 1$. La expresión $a \mod n$ denota el **único** $r \in \{0, \ldots, n-1\}$ tal que $n|(a-r)$*[1].

Presta especial atención al hecho de que $a \mod n$ siempre es un número no negativo, incluso si a es negativo. Una buena manera de recordar cómo funciona esto es:

a es igual a un múltiplo de n más $(a \mod n)$.

Ejemplo 1.2.
$21 \mod 7 = 0$ porque $21 = 3 \cdot 7 + 0$.

$20 \mod 7 = 6$ porque $20 = 2 \cdot 7 + 6$.

$-20 \mod 7 = 1$ porque $-20 = (-3) \cdot 7 + 1$.

$-1 \mod 7 = 6$ porque $-1 = (-1) \cdot 7 + 6$.

Definición 1.2.3. *Para $n \in \mathbb{N}$ positivo, escribiremos $\mathbb{Z}_n := \{0, \ldots, n-1\}$ para denotar el conjunto de **enteros módulo** n. Estos son los posibles restos que se obtienen al dividir por n.* [2]

[1] El hecho de que un único valor r tiene esta propiedad es un resultado estándar de matemática discreta.

[2] Los matemáticos protestarían con recelo ante esta definición por dos razones: (1) el hecho de que lo llamemos $\mathbb{Z}_n$ y no $\mathbb{Z}/(n\mathbb{Z})$; y (2) el hecho de que decimos que este conjunto contiene enteros en lugar de clases de congruencia de enteros. Ya que conoces la definición de clases de congruencia, podrás hacer la traducción mental con facilidad; y si no conoces la definición, no debería haber ningún caso en el que marque una diferencia.

Definición 1.2.4. *Para $n \in \mathbb{N}$ positivo, diremos que dos enteros a y b son* ***congruentes módulo*** *n, y escribiremos $a \equiv_n b$, si $n|(a-b)$. Alternativamente, $a \equiv_n b$ si y solo si $a \mod n = b \mod n$.*

Tendrás mejores posibilidades de tener éxito en esta asignatura si puedes comprender la (sutil) diferencia entre $a \equiv_n b$ y $a = b \mod n$.

$a \equiv_n b$:
$a = b \mod n$: La expresi

Ejemplo 1.3. "$99 \equiv_{10} 19$" es verdadero. Por definición, vemos que 10 divide a $99 - 19$.

Por otro lado, "$99 = 19 \mod 10$" es falso. Al hacer la operación del lado derecho, obtenemos el entero 9, pero 99 no es igual a 9.

En resumen, la expresión $a \equiv_n b$ tiene sentido para cualquier par de enteros a y b (incluyendo enteros negativos), mientras que la expresión $a = b \mod n$ sólo tiene sentido si $a \in \mathbb{Z}_n$.

La mayoría de los otros libros de texto usan la notación "$a \equiv b \mod n$" en lugar de "$a \equiv_n b$". No me gusta esa notación porque "mód n" se confunde fácilmente con una operación o acción que solo afecta al lado derecho, cuando en realidad es como un adverbio que modifica toda la expresión $a \equiv b$. Aunque $\equiv_n$ es un poco extraño, creo que vale la pena la rareza.

Si $d \mid x$ y $d \mid y$, entonces d es un **divisor común** de x e y. El mayor valor posible de d se llama el **máximo común divisor (MCD)**, y se denota como $\gcd(x, y)$. Si $\gcd(x, y) = 1$, decimos que x e y son **primos relativos**. El "algoritmo" más antiguo es el proceso recursivo que describió Euclides para calcular MCDs (aprox. 300 a.C.):

```
MCD(x, y): \\ Algoritmo de Euclides
If y = 0 then return x
else return MCD(y, x mód y)
```

Consejos y Trucos

A menudo te encontrarás con una expresión complicada y se te pedirá encontrar el valor de esa expresión módulo n. Esto generalmente significa: encuentra el valor único en $\mathbb{Z}_n$ que sea congruente con el resultado. La forma directa de hacerlo es calcular primero el resultado *sobre los enteros* y luego reducir la respuesta módulo n (es decir, con el operador

Si bien este método da la respuesta correcta (y es una buena base para tu comprensión), generalmente es aconsejable **simplificar los valores intermedios módulo** n a medida que avanzas. Hacerlo dará el mismo resultado final, pero normalmente será más fácil o rápido de calcular.

Ejemplo 1.4. Podemos evaluar la expresión $6 \cdot 7 \cdot 8 \cdot 9 \cdot 10 \mod 11$ sin calcular ese producto sobre los enteros usando el siguiente razonamiento:

$$\begin{aligned} 6 \cdot 7 \cdot 8 \cdot 9 &= (42) \cdot 8 \cdot 9 \cdot 10 \\ &\equiv_{11} 9 \cdot 8 \cdot 9 \cdot 10 \\ &= (72) \cdot 9 \cdot 10 \\ &\equiv_{11} 6 \cdot 9 \cdot 10 \\ &= (54) \cdot 10 \\ &\equiv_{11} 10 \cdot 10 \\ &= 100 \\ &\equiv_{11} 1. \end{aligned}$$

En los pasos que solo funcionan módulo 11, escribimos "$\equiv_{11}$". Escribimos "$=$" cuando el paso es válido en el conjunto de los enteros, aunque tampoco sería incorrecto usar "$\equiv_{11}$" en esos casos. Si dos expresiones representan el mismo número entero, entonces sin duda representan valores que son congruentes módulo 11.

Mi consejo es que simplifiques los valores intermedios módulo n, pero "simplificar" no siempre significa "reducir módulo n". A veces, una expresión puede "simplificarse" sustituyendo un valor por otro valor congruente, aunque no esté en el conjunto $\{0, \ldots, n-1\}$.

Ejemplo 1.5. Es posible calcular $7^{500} \mod 8$ mentalmente notando que $7 \equiv_8 -1$ y simplificando de la siguiente manera:

$$7^{500} \equiv_8 (-1)^{500} = 1.$$

De manera similar, es posible calcular $89^2 \mod 99$ mentalmente notando que $89 \equiv_{99} -10$, calculando:

$$89^2 \equiv_{99} (-10)^2 = 100 \equiv_{99} 1.$$

Puedes calcular cualquiera de estos resultados de la "forma larga" para verificar que estos atajos llevan a la respuesta correcta.

Dado que la suma, la resta y la multiplicación están definidas en el conjunto de los enteros (es decir, sumar/restar/multiplicar enteros siempre da como resultado otro entero), estos tipos de atajos pueden ser útiles.

Por otro lado, dividir enteros no siempre produce otro entero. ¿Tiene sentido usar la división cuando trabajamos módulo n, donde el resultado siempre debe estar en $\mathbb{Z}_n$? Responderemos esta pregunta más adelante en el manuscrito.

1.3. Cadenas de bits

Escribimos $\{0,1\}^n$ para denotar al conjunto de cadenas de n bits, y escribimos $\{0,1\}^*$ para denotar al conjunto de todas las cadenas de bits de longitud arbitraria pero finita. Si x es una cadena de bits, escribimos $|x|$ para denotar la longitud (o número de bits) de la cadena, y escribimos $\bar{x}$ para denotar el resultado de cambiar los 1's por 0's y 0's por 1's en x. Si el contexto es claro y sabemos que estamos hablando de cadenas de bits y no de números, escribimos 0^n y 1^n para denotar las cadenas de n ceros y n unos, respectivamente (no son la n-ésima potencia del entero 0 o 1).

Definición 1.3.1. *Cuando x e y son cadenas de la misma longitud, escribimos $x \oplus y$ para denotar la operación "or exclusivo" (*XOR*) bit a bit de las dos cadenas. La expresión $x \oplus y$ generalmente no está definida cuando las cadenas tienen longitudes diferentes, aunque en ocasiones raras puede ser útil considerar que la cadena más corta se completa con ceros. En ese caso, es necesario establecer una convención explícita sobre si la cadena más corta se rellena con ceros al inicio o al final.*

Por ejemplo, $0011 \oplus 0101 = 0110$. Las siguientes propiedades de la operación XOR serán de utilidad a lo largo del curso:

$x \oplus x = 000\cdots$	hacer el XOR de una cadena con sigo misma resulta en ceros.
$x \oplus 000\cdots = x$	el 0 es el neutro del XOR.
$x \oplus 111\cdots = \bar{x}$	hacer el XOR con una cadena de 1's cambia los 1's por 0's y 0's por 1's.
$x \oplus y = y \oplus x$	el operador XOR is conmutativo.
$(x \oplus y) \oplus z = x \oplus (y \oplus z)$	el operador XOR is asociativo.

Intenta usar estas propiedades para deducir la siguiente propiedad muy útil:

$$a = b \oplus c \quad \iff \quad b = a \oplus c \quad \iff \quad c = a \oplus b.$$

Existen algunas formas de entender el operador XOR que pueden ayudarte en esta asignatura:

- **Invertir bits:** Observa que hacer XOR con 0 no tiene ningún efecto, mientras que hacer XOR con 1 tiene como efecto invertir bits, es decir, cambia los 1's por 0's y 0's por 1's. Puedes pensar en $x \oplus y$ como: "partiendo de x, invierte los bits en todas las posiciones donde y tiene un 1." Entonces, si y está compuesto completamente de 1's, $x \oplus y$ invierte x bit a bit. Si $y = 1010\ldots$, entonces $x \oplus y$ significa "invertir el primer y tercer bit, de izquierda a derecha, en x."

  Muchos conceptos en esta asignatura pueden entenderse en términos de inversión de bits. Por ejemplo, podríamos preguntar "¿qué pasa si invierto el primer bit de x y lo envío a un algoritmo?" Esta pregunta también podría expresarse como "¿qué sucede si envío $x \oplus 1000\ldots$ al algoritmo?" Normalmente, no hay nada especial en invertir solo el primer bit de una cadena, así que a menudo es razonable generalizar la pregunta como "¿qué sucede cuando envío $x \oplus y$ al algoritmo, con una cadena y arbitraria (que no sea toda ceros)?"

- **Suma módulo 2:** El operador XOR es simplemente una suma módulo 2 bit a bit. Esta forma de pensar en XOR ayuda a explicar por qué algunas propiedades algebraicas como $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ son ciertas. Esta propiedades es válida para la suma, por lo que también lo es para el XOR.

  Esto también puede ayudarte a recordar por qué $x \oplus x$ resulta en una cadena de ceros. Si en lugar de XOR usáramos suma módulo 2, entonces vemos que $x + x = 2x$. Dado que $2 \equiv_2 0$, obtenemos que $2x \equiv_2 0x = 0$.

Definición 1.3.2. *Escribimos $x||y$ para denotar el resultado de concatenar x con y.*

1.4. Funciones

Sean X e Y dos conjuntos finitos. Una función $f : X \to Y$ es:

- **inyectiva:** si asocia entradas distintas con imágenes distintas. Formalmente, $x \neq x' \Rightarrow f(x) \neq f(x')$. Si existe una función inyectiva de X a Y, entonces necesariamente $|Y| \geq |X|$.
- **sobreyectiva:** si cada elemento de Y es la imagen de un elemento en X. Formalmente, para cada $y \in Y$, existe un $x \in X$ tal que $f(x) = y$. Si existe una función sobreyectiva de X a Y, entonces necesariamente $|Y| \leq |X|$.
- **biyectiva:** si f es inyectiva y sobreyectiva. Si existe una función biyectiva de X a Y, entonces necesariamente $|X| = |Y|$.

1.5. Probabilidad

Definición 1.5.1. *Una **distribución de probabilidad discreta** definida en un conjunto X de **sucesos** es una función, usualmente denotada por "*$\Pr$*", que asocia a cada suceso $x \in X$ una probabilidad* $\Pr[x]$*. Diremos que la distribución **asigna** la probabilidad* $\Pr[x]$ *al suceso x.*

Para cada suceso $x \in X$, la distribución de probabilidad debe satisfacer la condición $0 \leq \Pr[x] \leq 1$*. Adicionalmente, la suma de todas las probabilidades es 1:* $\sum_{x \in X} \Pr[x] = 1$.

Definición 1.5.2. *La **distribución uniforme** es la distribución de probabilidad tal que a cada $x \in X$ asigna la probabilidad* $\Pr[x] = 1/|X|$.

Extendemos la noción de Pr a **eventos**, es decir, conjuntos de sucesos. Formalmente, un evento A es un subconjunto de X, $A \subseteq X$, y su probabilidad se define como $\Pr[A] := \sum_{x \in A} \Pr[x]$. Además, con el fin de simplificar notación, en lugar de escribir $\Pr[\{x \mid x \text{ satisface cierta condición}\}]$ escribiremos $\Pr[\text{condición}]$.

Ejemplo 1.6. Considera un dado de 6 caras con cada cara identificada con un elemento del conjunto $X = \{1, 2, \ldots, 6\}$. Lanzar el dado (al menos para un matemático) induce una distribución uniforme sobre el conjunto X. Por tanto, $\Pr[3] = 1/6$, $\Pr[\text{número impar}] = 1/2$ y $\Pr[\text{número primo}] = 1/2$.

1.5.1. Consejos y trucos

Considera la probabilidad de que un evento A no ocurra, denotada por $\Pr[\neg A]$. Entonces es posible calcular la probabilidad de que A ocurra:

$$\Pr[A] = 1 - \Pr[\neg A].$$

Claramente, si conoces $\Pr[A]$ entonces prodrás calcular $\Pr[\neg A]$: $\Pr[\neg A] = 1 - \Pr[A]$. Esta relación es uno de los hechos más fundamentales de la probabilidad, pero puede llegar a ser sorprendentemente útil ya que a veces una de las dos probabilidades $\Pr[A]$ o $\Pr[\neg A]$ es más fácil de calcular que la otra. Así que si calcular $\Pr[A]$ parece una tarea inviable, entonces intenta calcular $\Pr[\neg A]$ en su lugar.

Ejemplo 1.7. Considera un dado de seis cara que se lanza seis veces. ¿Cuál es la probabilidad de que salga algún valor repetido?

Pensemos en todas las formas de obtener un valor repetido. Bien, podría ocurrir que dos de los lanzamientos den 1, o que tres de los lanzamientos den 1, o que todos resulten en 1, o que dos de ellos den 1 y los demás resulten en 2, etc. Pero... ¿estoy contando dos veces las repeticiones de 2 y las de 1?

Una manera más fácil de abordar este problema es darse cuenta de que la probabilidad que deseamos calcular es $1 - \Pr[\text{los seis lanzamientos dan valores distintos}]$. Este evento complementario (los seis lanzamientos dan valores distintos) ocurre precisamente cuando los valores que se obtienen en la secuencia de lanzamientos es una permutación del conjunto $\{1, \ldots, 6\}$. Hay $6! = 720$ permutaciones. El número total de sucesos es $6^6 = 46656$. Por tanto, la probabilidad de que salga algún valor repetido es

$$1 - \frac{6!}{6^6} = 1 - \frac{720}{46656} = \frac{45936}{46656} \approx 0{,}9846$$

Otro truco es introducir puntos de interrupción después de que haya ocurrido algunas elecciones aleatorias duranto un proceso, mientras que otras aún no han ocurrido. Este truco es mejor ilustrarlo con un ejemplo:

Ejemplo 1.8. Un problema clásico de probabilidad es el siguiente: al lanzar dos dados de seis caras, ¿cuál es la probabilidad de que ambos dados muestren el mismo número? Aquí tienes una manera estándar (y totalmente correcta) de responder a la pregunta:

Al lanzar dos dado

Una manera diferente de abordar la pregunta sería:

Considera lazar los

Otros trucos útiles son: (1) posponer una elección aleatoria hasta el último momento posible, justo antes de que su resultado se use por primera vez, y (2) cambiar el orden relativo de elecciones aleatorias independientes.

1.5.2. Terminología precisa

Es muy posible que en esta asignatura digamos cosas como "x es una cadena de bits aleatoria". Sin embargo, esta frase es imprecisa por muchas razones.

Primero, ¿es correcto decir que 42 es un número aleatorio? La aleatoriedad no es una propiedad de sucesos (un número o el resultado de lanzar una moneda). La aleatoriedad es una propiedad de un *proceso* que genera sucesos. Así que, en lugar de decir "x es una cadena de bits aleatoria", lo correcto es decir "x es una cadena de bits escogida de manera aleatoria".

Segundo, cuando usamos la palabra "aleatoriedad" nos referimos específicamente a la *distribución uniforme*. Así que, en lugar de decir "x es una cadena de bits aleatoria", lo correcto es decir "x es una cadena de bits escogida de manera uniformemente al azar" (suponiendo que se realmente se está usando la distribución uniforme).

Sin embargo, es muy común decir "x es una cadena de bits aleatoria" cuando en realidad queremos decir "x es una cadena de bits escogida de manera uniformemente al azar" siempre y cuando el contexto no dé lugar a confusión. Así que es importante que siempre recuerdes que **la aleatoriedad es una propiedad del proceso y no de los sucesos**.

Capítulo 2

Algoritmos y complejidad

Muchos problemas pueden resolverse considerándolos como casos especiales de un problema más general. Por ejemplo, encontrar el mayor de los números $101, 12, 144, 212, 98$ es un caso específico de localización del mayor valor de una sucesión de números enteros. Para resolver un problema como éste debemos dar un algoritmo que especifique una sucesión de pasos que conduzcan a una solución en el caso general. En este capítulo estudiaremos algoritmos para resolver muchos tipos de problemas diferentes. Por ejemplo, se desarrollarán algoritmos para encontrar el máximo común divisor de dos enteros, generar todas las ordenaciones de un conjunto finito, para búsqueda en listas o para ordenar elementos de una sucesión.

Una consideración importante en relación con un algoritmo es su complejidad computacional. Esto es, qué recursos computacionales se necesitan para usar este algoritmo en la resolución de un problema de un tamaño específico. Para medir la complejidad de un algoritmo utilizaremos las notaciones O y Θ, que presentaremos en este capítulo. También ilustraremos el análisis de la complejidad de algoritmos. Discutiremos el significado de la complejidad de un algoritmo en términos prácticos y teóricos.

El conjunto de los números enteros desempeña un papel fundamental en criptografía. En particular, el concepto de división de enteros es fundamental en aritmética computacional. Repasaremos brevemente algunos de los conceptos importantes en teoría de números, el estudio de los enteros y sus propiedades. Estudiaremos algunos algoritmos importantes relacionados con los enteros, entre los que se incluyen el algoritmo de Euclides para calcular el máximo común divisor, que fue descrito por primera vez hace miles de años. Los enteros se pueden representar usando cualquier entero positivo mayor que 1 como base. Las expresiones binarias, que se utilizan con mucha frecuencia en ciencias de la computación, se representan en base 2. Discutiremos en este capítulo las representaciones de enteros en base b y proporcionaremos algoritmos para encontrar estas representaciones. También describiremos los algoritmos para aritmética entera, que fueron los procedimientos llamados algoritmos. Asimismo, se presentan varias aplicaciones importantes de la teoría de números. Por ejemplo, utilizaremos la teoría de números para construir mensajes secretos, generar números pseudoaleatorios y asignar posiciones de memoria a ficheros de ordenador. La teoría de números, considerada

hace tiempo como las más pura de las disciplinas matemáticas, se ha convertido en una herramienta esencial para proporcionar seguridad en computación y en Internet.

2.1. Algoritmos

En matemática discreta aparecen muchas clases de problemas genéricos. Por ejemplo, dada una sucesión de números enteros, encontrar el mayor; dado un conjunto, enumerar todos sus subconjuntos; dado un conjunto de enteros, ponerlos en orden creciente; dada una red de ordenadores, encontrar el camino más corto entre dos nodos. Cuando se presentan tales problemas, lo primero que debemos hacer es construir un modelo que traduzca el problema a un contexto matemático. Las estructuras discretas utilizadas en estos modelo incluyen conjuntos, sucesiones y funciones, así como otras estructuras como permutaciones, relaciones, grafos, árboles, redes y máquinas de estados finitos.

Fijar el modelo matemático apropiado es solamente una parte de la solución. Para completarla se necesita un método que resuelva el problema general utilizando el modelo. Idealmente, lo que se requiere es un procedimiento que siga una secuencia de pasos que conduzca a la repuesta deseada. Tal secuencia de pasos se denomina **algoritmo**.

Definición 2.1.1. *Un algoritmo es un conjunto finito de instrucciones precisas que sirve para realizar un cálculo o resolver un problema.*

En este capítulo discutiremos algoritmos que resuelven una gran variedad de problemas. En esta sección usaremos el problema de encontrar el mayor de una sucesión finita de enteros para ilustrar el concepto de algoritmo y sus propiedades. También describiremos algoritmos para localizar un elemento particular de un conjunto. En secciones posteriores se describirán, entre otros, procedimientos para hallar el máximo común divisor de dos enteros, para hallar el recorrido más corto entre dos puntos de una red o para multiplicar matrices.

Ejemplo 2.1. Describe un algoritmo para encontrar el máximo valor de una sucesión finita de enteros.

Aunque encontrar el mayor de una sucesión de enteros es relativamente trivial, este problema proporciona una buena ilustración del concepto de algoritmo. Además, hay muchos casos en los que se requiera el mayor de los enteros de una sucesión finita. Por ejemplo, una universidad puede necesitar encontrar al estudiante con mayor puntuación entre los miles que han realizado un examen. O una organización deportiva puede querer identificar cada mes al participante con mayor puntuación. Queremos desarrollar un algoritmo que se pueda usar siempre que aparezca un problema de búsqueda del mayor elemento de una sucesión finita de enteros.

Podemos especificar un procedimiento para resolver este problema de varias formas. Un método consiste simplemente en utilizar el lenguaje natural para describir la sucesión de pasos utilizada. Ésta es la solución que proporciona este método:

Solución: Realizamos los siguientes pasos.

1. Fijamos provisionalmente el máximo igual al primer elemento de la sucesión. (El máximo provisional será el mayor entero examinado hasta esa etapa del procedimiento).

2. Comparamos el siguiente entero de la sucesión con el máximo provisional, y si es más grande que el máximo provisional, fijamos el máximo provisional como este entero.
3. Se repite el paso anterior si hay más enteros de la sucesión.
4. Paramos cuando no queden más enteros en la sucesión por comparar. Llegado este punto, el máximo provisional es el mayor entero de la sucesión.

2.2. Notación de pseudocódigo

Un algoritmo se puede describir empleando un lenguaje informático. Cuando se usa un lenguaje informático, sólo podemos utilizar para describir el algoritmo las instrucciones permitidas en ese lenguaje particular. Esto a veces conduce a una descripción del algoritmo difícil de entender. Como comúnmente se utilizan varios lenguajes de programación distintos en vez de usar uno en particular para especificar algoritmos, en esta asignatura utilizaremos una forma de **pseudocódigo**. (Todos los algoritmos se describirán también en lenguaje natural.) El pseudocódigo proporciona un paso intermedio entre una descripción de este algoritmo en lenguaje natural y en lenguaje de programación. Los pasos del algoritmo se especifican usando instrucciones que nos recordarán a aquellas que se utilizan en programación. No obstante, en pseudocódigo, las instrucciones utilizadas pueden incluir cualquier operación o sentencia bien definida. Un programa informático se puede generar en cualquier lenguaje de programación utilizando como punto de partida una descripción en psuedocódigo.

```
MAX(a_1, a_2, ..., a_n): \\ Búsqueda del máximo
max:= a_1
for i := 2 to n
    if max< a_i then max:= a_i
end
return max
```

Este algoritmo primero asigna el término inicial de la sucesión a_1 a la variable max. El bucle "for" se usa para examinar sucesivamente los términos de la sucesión. Si un término a_i es mayor que el valor de max en ese momento, se asigna éste como nuevo valor de max.

Hay varias propiedades que generalmente comparten los algoritmos. Es útil tener estas propiedades en mente cuando se describen algoritmos. Estas propiedades son:

- **Entrada.** Un algoritmo tiene valores de entrada que son elementos de un conjunto especificado.
- **Salida.** Para cada conjunto de valores de entrada, un algoritmo produce valores de salida de un conjunto especificado. Los valores de salida son la solución del problema.
- **Definición.** Los pasos de un algoritmo deben definirse con precisión.
- **Corrección.** Un algoritmo debe producir valores de salida correctos para cada conjunto de valores de entrada.
- **Duración finita.** Un algoritmo debe producir la salida deseada tras un número finito (aunque quizá grande) de pasos para cualquier conjunto de valores de entrada.

- **Efectividad.** Debe ser posible realizar cada paso del algoritmo con exactitud y en un intervalo finito de tiempo.
- **Generalidad.** El procedimiento debería ser aplicable a todos los problemas de la forma deseada, no sólo para un conjunto particular de datos de entrada.

Ejemplo 2.2. Muestra que el algoritmo de búsqueda del máximo elemento de una sucesión finita de enteros tiene todas las propiedades enumeradas previamente.

Solución: La entrada del algoritmo de búsqueda es una sucesión finita de enteros. La salida es el mayor entero de la sucesión. Cada paso del algoritmo está definido con precisión, puesto que sólo son asignaciones, un bucle finito y una sentencia condicional. Para mostrar que el algoritmo es correcto, debemos mostrar que cuando el algoritmo concluye, el valor de la variable max coincide con el máximo de los términos de la sucesión. Para ver esto, observa que el valor inicial de max es el primer término de la sucesión; cuando se examina sucesivamente cada término, el valor de max se actualiza si el término es mayor que el máximo de los términos anteriormente examinados. Este argumento (informal) muestra que cuando se hayan examinado todos los términos, max coincide con el mayor valor. El algoritmo realiza un número finito de pasos, puesto que termina cuando se han examinado todos los números de la sucesión. El algoritmo puede desarrollarse en un tiempo finito, puesto que cada paso es bien una comparación o una asignación. Finalmente, el algoritmo es general, puesto que puede emplearse para encontrar el máximo de cualquier sucesión finita de números enteros.

Además, usaremos la siguiente notación:

$\leftarrow$ Cuando $\mathcal{D}$ es una dist

$:=$

$\stackrel{?}{=}$

2.3. Algoritmos de búsqueda

El problema de localizar un elemento de una lista ordenada se puede encontrar en muchos contextos. Por ejemplo, un programa que revisa que las palabras de un texto estén correctamente escritas busca estas palabras en un diccionario, que no es más que una lista ordenada de palabras. Los problemas de este tipo se llaman **problemas de búsqueda**. En esta sección describiremos varios algoritmos de búsqueda.

Un problema de búsqueda general se puede describir como sigue: localizar un elemento x en una lista de elementos distintos $a_1, a_2, \ldots, a_n$ o determinar que no está en la lista. La solución a este problema de búsqueda es la localización del término en la lista que es igual que x (esto es, la solución es i si $x = a_i$) y es 0 si x no está en la lista.

2.3.1. Búsqueda lineal

El primer algoritmo que presentaremos es el conocido como **búsqueda lineal** o **búsqueda secuencial**. El algoritmo de búsqueda lineal comienza por comparar x y a_1. Cuando $x = a_1$, la solución es la localización

de a_1, es decir, 1. Cuando $x \neq a_1$, x se compara con a_2. Si $x = a_2$, la solución resulta ser la localización de a_2, es decir, 2. Cuando $x \neq a_2$, se compara x con a_3. Este proceso se continúa, comparando x sucesivamente con cada término de la lista hasta que se encuentra una coincidencia. La solución es la localización de este término, a no ser que esta coincidencia no ocurra. Si se ha recorrido la lista completa sin localizar x, la solución es 0. El pseudocódigo para el algoritmo de la búsqueda lineal se muestra en el siguiente algoritmo.

```
LINSEARCH(x; a_1, a_2, ..., a_n): \\ Búsqueda lineal
i := 1
b := false
While (i ≤ n y b = false)
    i := i + 1
    b := (x ?= a_i)
end
if i ≤ n then localización := i
else localización:= 0
return localización
```

2.3.2. Búsqueda binaria

Consideramos ahora otro algoritmo de búsqueda. Este algoritmo se puede usar cuando la lista tiene los términos en orden creciente de tamaño (por ejemplo, si los términos son números, se listan de menor a mayor; si son palabras, se listan en orden lexicográfico o alfabético). Este segundo algoritmo se llama *algoritmo de búsqueda binaria*. Se desarrolla comparando el elemento que se quiere localizar con el elemento central de la lista. La lista entonces se parte en dos sublistas más pequeñas. Ambas tienen el mismo tamaño a una de ellas tiene un elemento más que la otra. La búsqueda continúa restringiéndose a la lista apropiada, basándose en la comparación del elemento que se desea localizar con el término central. Más adelante se mostrará que el algoritmo de búsqueda binaria es mucho más eficiente que el de búsqueda lineal. El siguiente ejemplo muestra cómo se realiza una búsqueda binaria.

Ejemplo 2.3. Para buscar el número 19 en la lista

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22

primero partimos en la lista, que tiene 16 términos, en dos más con 8 términos cada una, es decir

1 2 3 5 6 7 8 10 12 13 15 16 18 19 20 22.

Entonces, 19 se compara con el mayor término de la primera sublista. Como $10 < 19$, la búsqueda para 19 se puede restringir a la lista que contiene desde el término noveno al decimosexto de la lista original. A continuación dividimos esta lista, que tiene ocho términos, en dos sublistas de cuatro elementos cada una, a saber,

12 13 15 16 18 19 20 22.

Como $16 < 19$ (comparando 19 con el mayor elemento de la primera lista), la búsqueda se restringe a la segunda de estas listas, la que contiene desde el término decimotercero al decimosexto de la lista original. Esta lista, 18 19 20 22, se parte de nuevo en otras dos,

18 19 20 22.

Como 19 no es mayor que el mayor término de la primera de estas dos listas, el cual también es 19, la búsqueda se restringe a la primera lista, 18 19, que contiene los términos decimotercero y decimocuarto de la lista original. Esta lista de nuevo se parte en dos listas, de un término cada una: 18 y 19. Como $18 < 19$, la búsqueda se restringe a la segunda lista, que contiene el término decimocuarto de la lista original, que es 19. Ahora que la búsqueda se ha reducido a una lista de un término, se hace una comparación, y localizamos el valor 19 en la posición decimocuarta de la lista.

El nivel de detalle del ejemplo anterior nos puede parecer excesivo, pero debemos recordar que nuestro objetivo es describir un algoritmo que funcione para *cualquier lista ordenada*. Para ello es necesario establecer un algoritmo con pasos bien definidos y detallados como hemos hecho en el ejemplo.

Ahora especificaremos los pasos del algoritmo de búsqueda binaria. Para buscar el entero x en la lista $a_1, a_2, \ldots, a_n$, donde $a_1 < a_2 < \cdots < a_n$, comenzamos comparando x con el término a_m central de la lista, donde $m = \lfloor (n+1)/2 \rfloor$. (Recuerda que $\lfloor x \rfloor$ es el mayor entero que es menor o igual que x). Si $x > a_m$, la búsqueda de x se puede restringir a la segunda mitad de la lista, que es $a_{m+1}, a_{m+2}, \ldots, a_n$. Si x no es mayor que a_m, la búsqueda de x se puede restringir a la primera mitad de la lista, que es $a_1, a_2, \ldots, a_m$.

La búsqueda se ha restingido a una sublista con no más de $\lceil n/2 \rceil$ elementos ($\lceil x \rceil$ es el menor entero mayor o igual que x). Utilizando el mismo procedimiento, se compara x con el término del medio de la sublista a la que hemos restringido la búsqueda. Entonces, la búsqueda se vuelve a restringir a una de las dos mitades de esta sublista. Este proceso se repite hasta que se obtiene una sublista con un único término. Entonces se determina si este término es x. El pseudocódigo para el algoritmo de búsqueda binaria se muestra a continuación.

```
BINSEARCH(x; a_1 < a_2 < ··· < a_n): \\ Búsqueda binaria
i := 1\\ i es el extremo izquierdo del intervalo de búsqueda
j := n\\ j es el extremo derecho del intervalo de búsqueda
While (i ≤ j)
    m := ⌊(i + j)/2⌋
    if x > a_m then i := m + 1
    else j := m
end
if x = a_i then localización := i
else localización:= 0
return localización
```

Este algoritmo se desarrolla reduciendo sucesivamente el rango de la lista en la que se busca. En una posición determinada, sólo se consideran los términos que empiezan por a_i y terminan en a_j. En otras palabras, i y j son los subíndices menor y mayor respectivamente de los términos en los que se busca. El algoritmo continúa reduciendo la parte de la lista sobre la que se busca hasta que se llega a una lista de un solo término. Cuando se llega a este punto, se hace una comparación para ver si este término coincide con x.

2.4. Algoritmos de ordenación

Ordenar los elementos de una lista es un problema que se presenta en muchos contextos. Por ejemplo, para generar un directorio telefónico es necesario ordenar alfabéticamente los nombres de los usuarios. Poner en orden las direcciones en una lista de envíos de mensajes de correo electrónico puede determinar si hay direcciones duplicadas. Crear un diccionario requiere que las palabras se pongan en orden alfabético. De forma similar, para generar un listado útil de objetos ordenamos en orden creciente de referencias.

Supongamos que tenemos una lista de elementos de un conjunto. Además, supongamos que conocemos una forma de ordenar estos elementos. Una **ordenación** es colocar estos elementos en una lista en la cual los elementos se disponen en orden creciente. Por ejemplo, ordenar la lista $7, 2, 1, 4, 5, 9$ produce la lista $1, 2, 4, 5, 7, 9$. Ordenar la lista *d,h,c,a,f* (usando el orden alfabético) da lugar a la lista *a,c,d,f,h*.

En esta sección presentaremos dos algoritmos de ordenación: el método de la burbuja y la ordenación por inserción.

2.4.1. Método de la burbuja

La **ordenación de burbuja** es uno de los algoritmos de búsqueda más simples, pero no uno de los más eficientes. Coloca los elementos de una lista en orden creciente mediante la comparación sucesiva de elementos adyacentes, intercambiándolos si están en el orden equivocado. Para desarrollar el método de la burbuja, llevamos a cabo la operación básica, intercambiar un elemento mayor por otro menor que le siga, comenzando por el principio de la lista hasta llegar al final. Iteramos este proceso hasta que la ordenación se completa. Un algoritmo en pseudocódigo para la ordenación de burbuja se da a continuación.

```
BUBBLE(a_1, a_2, ··· , a_n): \\ La ordenación de burbuja
for i := 1 to n − 1
    for j := 1 to n − i
        if a_j > a_{j+1} then
          m := a_j
          a_j := a_{j+1}
          a_{j+1} := m
    end
end
return a_1, a_2, ..., a_n \\ Lista en orden creciente
```

Podemos imaginar los elementos de la lista ordenada en una columna. En el ordenamiento de la burbuja, el menor elemento "flota" hacia arriba mientras es intercambiado por elementos mayores. Los elementos mayores se "hunden" al fondo. Esto se ilustra en el siguiente ejemplo.

Ejemplo 2.4. Usa la ordenación de burbuja para poner $3, 2, 4, 1, 5$ en orden creciente.

Solución: Los pasos de este algoritmo se ilustran en la Figura 2.1. Se comienza comparando los dos primeros elementos, 3 y 2. Como $3 > 2$, se intercambian 3 y 2, produciendo la lista $2, 3, 4, 1, 5$. Como $3 < 4$, se continúa comparando 4 y 1. Como $4 > 1$, se intercambian 1 y 4, produciendo la lista $2, 3, 1, 4, 5$. Como

$4 < 5$, se completa la primera pasada. La primera pasada garantiza que el mayor elemento, 5, está en la posición correcta.

La segunda pasada comienza comparando 2 y 3. Como están en orden correcto, se comparan 3 y 1. Como $3 > 1$, estos números se intercambian, produciendo la lista $2, 1, 3, 4, 5$. Como $3 < 4$, éstos están en el orden correcto. No es necesario hacer más comparaciones, puesto que 5 ya está en la posición correcta. La segunda pasada garantiza que los dos elementos, 4 y 5, están en las posiciones correctas.

La tercera pasada comienza comparando 2 y 1. Éstos se intercambian, ya que $2 > 1$, produciendo la lista $1, 2, 3, 4, 5$. Como $2 < 3$, estos dos elementos están en el orden correcto. No es necesario hacer más comparaciones en esta pasada por que 4 y 5 ya están en sus posiciones correctas. La tercera pasada garantiza que los tres mayores elementos $3, 4$ y 5, están en sus posiciones correctas.

La cuarta pasada consiste en una comparación, la comparación de 1 y 2. Como $1 < 2$, estos elementos están en el orden correcto. Esto completa la ordenación de burbuja.

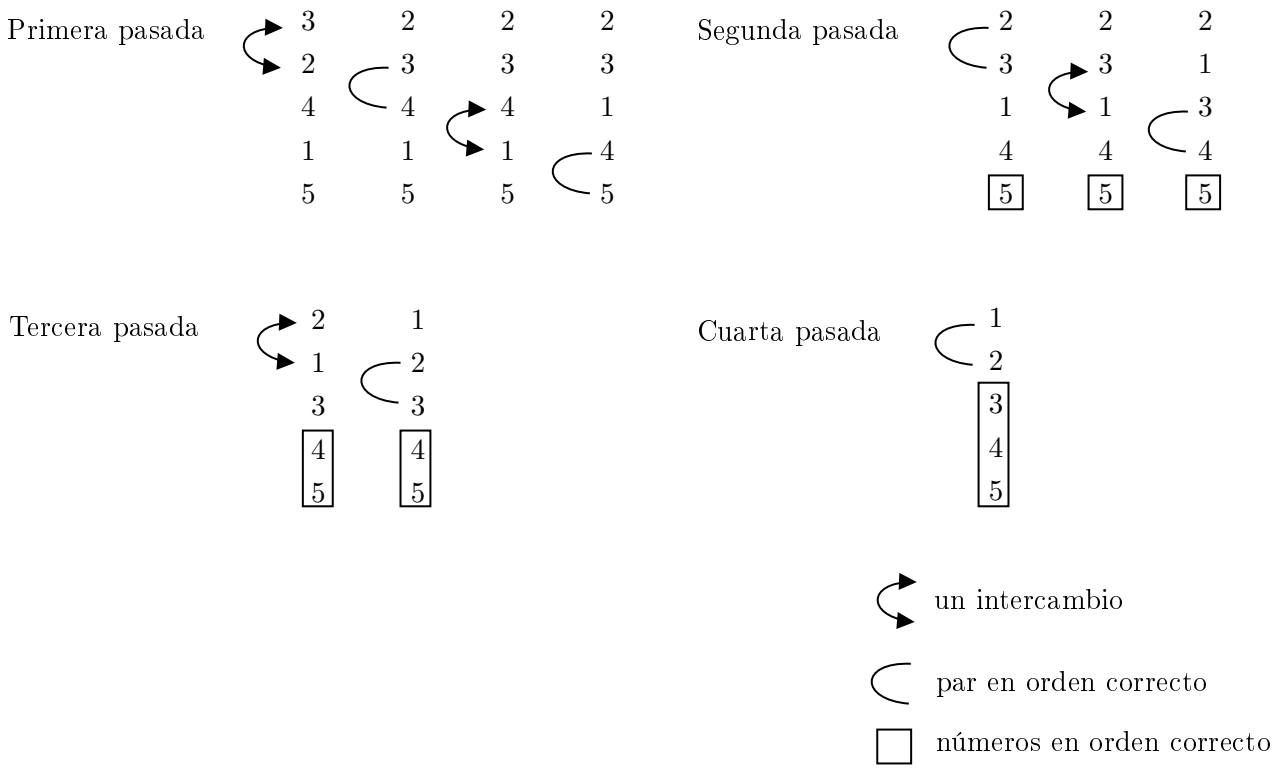


Figura 2.1: Los pasos de una ordenación por el método de burbuja.

2.4.2. Ordenación por inserción

La **ordenación por inserción** es un algoritmo de ordenación simple, pero no es generalmente el más eficiente. Para ordenar una lista con n elementos, la ordenación por inserción comienza con el segundo elemento. Se compara este segundo elemento con el primero y se coloca antes del primero si no es mayor que

el primer elemento, y tras el primer elemento si es mayor que éste. En este punto, los dos primeros elementos están en el orden correcto. El tercer elemento se compara con el primero, y si es mayor que él, se compara con el segundo. Se coloca en la posición correcta entre los primeros tres elementos.

En general, en el paso j-ésimo de la ordenación por inserción, el elemento j-ésimo de la lista se inserta en la posición correcta de la lista formada por los $j-1$ elementos previamente ordenados. Para insertar el elemento j-ésimo en la lista, se utiliza una búsqueda lineal: el elemento j-ésimo se compara con los $j-1$ del comienzo de la lista ya ordenados, hasta que se encuentra el primer elemento que no es menor que éste o hasta que se haya comparado con los primeros $j-1$ elementos. El elemento j-ésimo se coloca en la posición correcta, por lo que los primeros j elementos quedan ordenados. El algoritmo continúa hasta que el último elemento se inserta en la posición correcta con respecto a los $n-1$ primeros elementos ya ordenados. La ordenación por inserción se describe en pseudocódigo a continuación.

```
INSERT(a_1, a_2, ··· , a_n): \\ Ordenación por inserción
for j := 2 to n
    i := 1
    While (a_j > a_i)
       i := i + 1
    end
    m := a_j
    for k := 0 to j − i − 1
       a_{j−k} := a_{j−k−1}
    end
    a_i := m
end
return a_1, a_2, ..., a_n \\ Lista en orden creciente
```

Ejemplo 2.5. Utiliza la ordenación por inserción para poner la lista $3, 2, 4, 1, 5$ en orden creciente.

Solución: La ordenación por inserción compara primero 2 y 3. Como $3 > 2$, coloca 2 en la primera posición produciendo la lista $\mathbf{2}, \mathbf{3}, 4, 1, 5$ (la parte ordenada de la lista se muestra en negrita). En este punto, 2 y 3 están en orden correcto. Posteriormente, se inserta el tercer elemento, 4, en la parte ya ordenada de la lista mediante las comparaciones $4 > 2$ y $4 > 3$. Como $4 > 3$, 4 se coloca en la tercera posición. En este punto, la lista $\mathbf{2}, \mathbf{3}, \mathbf{4}, 1, 5$ y sabemos que el orden de los tres primeros elementos es correcto. Ahora buscamos la posición correcta para el cuarto elemento, 1, entre los elementos ya ordenados $2, 3, 4$. Como $1 < 2$, obtenemos la lista $\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, 5$. Finalmente, colocamos 5 en la posición correcta por comparaciones sucesivas con $1, 2, 3$ y 4. Como $5 > 4$, va al final de la lista, produciendo el orden correcto para la lista completa.

2.5. Algoritmos voraces

Muchos algoritmos están diseñados para resolver **problemas de optimización**. El objetivos de tales problemas es encontrar la solución al problema dado que bien minimice o maximice el valor de algún parámetro.

Sorprendentemente, una técnica muy simple conduce a menudo una solución óptima del problema. Esta técnica hace la mejor elección en cada paso en vez de considerar toda la secuencia global de pasos que podría conducir a una solución óptima. Los algoritmos que hacen lo que parece ser la "mejor" elección en cada paso se conocen como **algoritmos voraces** (*greedy* en terminología inglesa). Una vez conocida la solución proporcionada por un algoritmo voraz, tenemos que determinar si es óptima. Para hacer esto, bien demostraremos que la solución es realmente óptima o bien presentamos un contraejemplo en el que el algoritmo conduce a una solución que no es óptima. Para concretar estos conceptos, consideraremos un algoritmo para dar cambio de monedas.

Ejemplo 2.6. Consideramos el problema de dar cambio de n céntimos en monedas de $1, 5, 10$ y 25 céntimos usando el mínimo número de monedas posibles (aunque no existan monedas de 25 céntimos de Euro,usaremos estos valores para ilustrar el concepto de algoritmo voraz; además en otros países sí existen monedas de 25 céntimos, por ejemplo, el dólar americano tiene monedas de 25 céntimos). Podemos inventar un algoritmo voraz para dar cambio de n céntimos haciendo optimizaciones locales en cada paso, esto es, en cada paso escogemos la moneda de mayor valor posible que no exceda el valor n y la añadimos al monto del cambio. Por ejemplo, para dar cambio de 67 céntimos seleccionamos una moneda de 25 céntimos (dejando 42 céntimos). Luego seleccionamos otra de 25 céntimos (quedando 17), seguimos con una de 10 (y quedan 7). Seguimos con una de 5 (quedan 2 céntimos), seguimos con una moneda de 1 (quedando 1 céntimo) y acabamos con una moneda de 1.

El siguiente algoritmo es un algoritmo voraz de cambio de n céntimos utilizando cualquier conjunto de valores de monedas. Toma como entrada una lista de valores de las monedas $c_1, c_2, \ldots, c_r$ tal que $c_1 > c_2 > \cdots > c_r$, y un valor entero positivo n.

```
CHANGE(c_1 > c_2 > ··· > c_r; n): \\ Algoritmo voraz para dar cambio
cambio:=[ ] \\ lista vacía
for i := 1 to r
      While (n ≥ c_i)
         cambio:=[cambio, moneda de valor c_i] \\Se añade una moneda de valor c_i al cambio
         n := n − c_i
      end
end
return cambio
```

Hemos descrito un algoritmo voraz para dar cambio válido para monedas de $1, 5, 10$ y 25 céntimos. Mostraremos que este algoritmo conduce a una solución óptima para este conjunto de monedas en el sentido de que da el menor número de monedas posibles. Antes de presentar la demostración, veamos que hay otros conjuntos de monedas para las cuales el algoritmo voraz presentado no produce necesariamente el cambio con el mínimo de monedas posibles (hemos dicho que un algoritmo voraz hace lo que parece ser la "mejor" elección **en cada paso**, pero en ningún momento hemos dicho que garantiza una solución óptima para cualquier entrada). Por ejemplo, supongamos que sólo tenemos monedas de $25, 10$ y 1 céntimos (no de 5 céntimos). Para 30 céntimos, el algoritmo voraz daría el cambio usando seis monedas – una moneda de 25 y cinco de 1 –, mientras que podríamos haber usado tres monedas utilizando las de 10 céntimos.

Lema 2.5.1. *Si n es un entero positivo, entonces el cambio de n céntimos con monedas de 25, 10, 5 y 1 céntimos utilizando el menor número posible de monedas tiene como máximo dos de 10 , como máximo una de 5 y como máximo cuatro de 1, y no puede tener dos de 10 y una de 5. El monto del cambio en monedas de $10, 5$ y 1 céntimo no puede exceder 24 céntimos.*

Demostración. Daremos una demostración por reducción al absurdo. Supongamos que, al dar cambio de n céntimos utilizando el menor número de monedas, es posible tener más monedas de cada tipo que las especificadas en el enunciado del lema. Mostraremos que, bajo esta suposición, podríamos reemplazar dichas monedas por un menor número de monedas con el mismo valor total, lo cual contradice el hecho de que estamos usando el menor número de monedas.

Analicemos cada caso:

1. **Monedas de 10 céntimos:** Si tuviésemos tres monedas de 10 céntimos, podríamos reemplazarlas por una moneda de 25 céntimos y una de 5 céntimos, reduciendo así el número total de monedas.
2. **Monedas de 5 céntimos:** Si tuviésemos dos monedas de 5 céntimos, podríamos reemplazarlas por una moneda de 10 céntimos, nuevamente reduciendo el número total de monedas.
3. **Combinación de monedas de 10 y 5:** Si tuviésemos dos monedas de 10 céntimos y una de 5 céntimos, podríamos reemplazarlas por una moneda de 25 céntimos, logrando una solución con menos monedas.

Dado lo anterior, concluimos que es imposible tener más de dos monedas de 10 céntimos, más de una moneda de 5 céntimos, o una combinación de dos monedas de 10 y una de 5 céntimos al mismo tiempo.

Por lo tanto, la cantidad máxima de dinero que podemos reunir utilizando monedas de 10, 5 y 1 céntimos sin exceder el número mínimo de monedas es 24 céntimos, cumpliendo las siguientes restricciones: como máximo dos monedas de 10, como máximo una moneda de 5, y como máximo cuatro monedas de 1. ■

Teorema 2.5.2. *El algoritmo voraz para dar cambio con monedas de $25, 10, 5$ y 1 céntimos, lo hace con el menor número de monedas posibles.*

Demostración. Demostraremos el teorema por reducción al absurdo. Supongamos que existe un entero positivo n tal que existe una forma de dar cambio de n céntimos utilizando monedas de 25, 10, 5 y 1 céntimos que emplee menos monedas que el algoritmo voraz.

Primero, consideremos el número de monedas de 25, q', que se utilizan en esta supuesta solución óptima. Observaremos que q' debe ser igual al número de monedas de 25, q, que emplea el algoritmo voraz. Para demostrarlo:

I) **Caso $q' \leq q$:** Dado que el algoritmo voraz utiliza el máximo número posible de monedas de 25 sin exceder n, se sigue que $q' \leq q$.

II) **Caso $q' < q$:** Si q' fuera menor que q, entonces la cantidad restante por cubrir después de usar q' monedas de 25 céntimos sería al menos 25 céntimos. Esto implicaría que sería necesario reunir esos 25 céntimos restantes usando únicamente monedas de 10, 5 y 1 céntimos. Sin embargo, esto es imposible por el Lema 2.5.1, ya que el monto total en monedas de 10, 5 y 1 no puede exceder 24 céntimos.

Por lo tanto, debemos tener $q' = q$.

Dado que el número de monedas de 25 es el mismo en ambos métodos, el monto restante a cubrir utilizando monedas de 10, 5 y 1 céntimos también será el mismo en ambos casos y no excederá 24 céntimos.

A continuación, analizamos las monedas de 10, 5 y 1:

1. **Monedas de 10:** El algoritmo voraz utiliza el máximo número posible de monedas de 10, respetando la restricción de no exceder el monto restante. Por el Lema 2.5.1, cuando se busca minimizar el número total de monedas, también se utiliza el mayor número posible de monedas de 10. Por lo tanto, en la solución óptima también debe haber el mismo número de monedas de 10.
2. **Monedas de 5:** Por el Lema 2.5.1, en la solución óptima como máximo se utiliza una moneda de 5. El algoritmo voraz también sigue esta restricción, por lo que ambos métodos emplean el mismo número de monedas de 5.
3. **Monedas de 1:** Finalmente, por el Lema 2.5.1, como máximo se pueden usar cuatro monedas de 1. El algoritmo voraz utiliza el mínimo número posible de monedas de 1, y lo mismo ocurre en la solución óptima, por lo que ambos métodos coinciden también en esta cantidad.

Dado que el número de monedas de cada denominación es el mismo en ambos métodos, llegamos a una contradicción con nuestra suposición inicial de que la solución óptima utiliza menos monedas que el algoritmo voraz. Por lo tanto, concluimos que el algoritmo voraz proporciona efectivamente el cambio con el menor número posible de monedas. ■

Es importante hacer la siguiente reflexión. Hemos mencionado anteriormente que, si únicamente disponemos de monedas de 25, 10 y 1 céntimos (es decir, no contamos con monedas de 5 céntimos), el algoritmo voraz para dar cambio no proporciona una solución óptima en el caso de que se deba devolver un cambio de 30 céntimos. Sin embargo, esta observación no contradice el enunciado del Teorema 2.5.2. El teorema se aplica exclusivamente al caso particular en el que disponemos de monedas de 25, 10, 5 y 1 céntimos, y no hace ninguna afirmación sobre otros conjuntos de denominaciones. Por tanto, es importante destacar que los resultados del teorema no pueden generalizarse a escenarios con un sistema de monedas diferente y que la validez del teorema está intrínsecamente ligada a las condiciones iniciales establecidas.

2.6. Crecimiento de funciones

En la sección anterior, discutimos el concepto de algoritmo y presentamos algunos ejemplos que resuelven problemas como la búsqueda de un elemento en una lista o la ordenación de una lista. Más adelante, estudiaremos cómo analizar el número de operaciones que realizan estos algoritmos para resolver un problema. En particular, estimaremos el número de comparaciones que los algoritmos de búsqueda lineal y búsqueda binaria necesitan para localizar un elemento en una sucesión de n elementos. Asimismo, calcularemos el número de comparaciones realizadas por los métodos de ordenación por burbuja e inserción al ordenar una lista de n elementos.

El tiempo requerido para resolver un problema no depende únicamente del número de operaciones realizadas por un algoritmo. También está influido por el *hardware* y el *software* empleados para ejecutar el

programa que implementa dicho algoritmo. Sin embargo, al cambiar de plataforma, el tiempo de ejecución para un problema de tamaño n se puede aproximar razonablemente bien mediante una constante multiplicativa. Por ejemplo, un supercomputador debería ser capaz de resolver un problema de tamaño n en un tiempo aproximadamente un millón de veces menor que un PC convencional. Sin embargo, este factor de aceleración no depende de n.

Una de las ventajas clave de la **notación** O, que introduciremos en esta sección, es que nos permite analizar el crecimiento de una función ignorando las constantes multiplicativas y los términos de menor orden. Esto significa que, al usar la notación O, no es necesario preocuparnos por el *hardware* o el *software* específico utilizado en la implementación del algoritmo. Además, podemos asumir que todas las operaciones dentro del algoritmo consumen el mismo tiempo, lo cual simplifica enormemente el análisis y nos permite centrarnos en los aspectos fundamentales del rendimiento del algoritmo.

La notación O es ampliamente utilizado para estimar el número de operaciones que realiza un algoritmo a medida que su entrada crece. Con la ayuda de esta notación podemos determinar si es práctico usar un algoritmo particular para solucionar un problema a medida que el tamaño de la entrada aumenta. Además, utilizando esta notación, podemos comparar dos algoritmos para determinar cuál es el más eficiente en función del crecimiento de la entrada. Por ejemplo, si tenemos dos algoritmos que resuelven un problema, uno usando $100n^2 + 17n + 4$ operaciones y otro n^3 operaciones, la notación O nos puede ayudar a ver que el primero de ellos realiza menos operaciones cuando n es grande, incluso aunque necesite más operaciones para valores pequeños, como $n = 10$.

Esta sección presenta la notación O y la relaciona con las notaciones Omega y Zeta. Explicaremos cómo se construyen las estimaciones O, Omega y Zeta y desarrollaremos estimaciones de algunas funciones importantes que se utilizan en el análisis de algoritmos.

2.6.1. La notación O mayúscula

El crecimiento de las funciones se describe a menudo mediante una notación especial, que se presenta a continuación.

Definición 2.6.1. *Sean $f, g : \mathbb{N} \to \mathbb{N}$ funciones positivas. Diremos que $f(n) = O(g(n))$ si y solo si*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$$

Equivalentemente, $f(n) = O(g(n))$ si y solo si para cada entero $N > 0$ existe una constante $c > 0$ tal que

$$si\ n > N entonces\ f(n) < c\,g(n).$$

Esto se puede leer como "f es o mayúscula de g" o "f es una o mayúscula de g".

Esta definición (y las siguientes) es *asintótica* porque es posible que la desigualdad anterior no se cumpla para valores de n más pequeños que N.

Las constantes c y N de la definición se llaman **testigos** de la relación $f(n) = O(g(n))$. Para establecer que $f(n)$ es $O(g(n))$ necesitamos encontrar sólo *un* para de constantes c y N, los testigos, tales que $f(n) < cg(n)$ siempre que $n > N$.

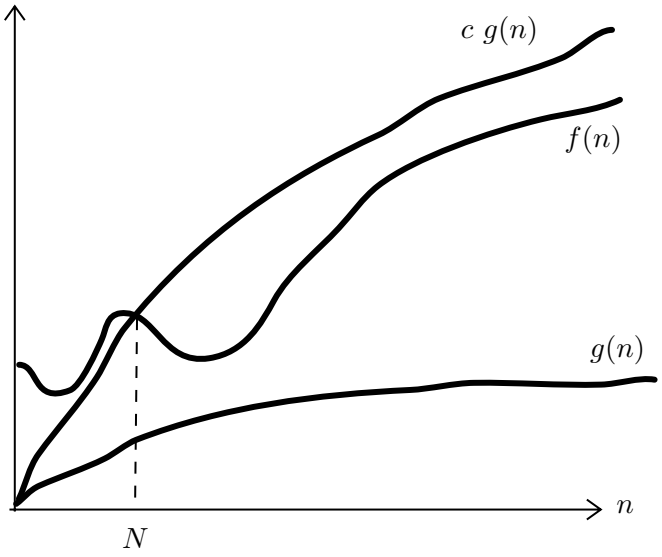


Figura 2.2: $f(n) = O(g(n))$.

Ten en uenta que si hay un par de testigos para esta relación, habrá *infinitos* pares de tetigos. Para ver esto, observa que si c y N son un par de testigos, entonces cualquier para d y M, donde $c < d$ y $N < M$, es también un par de testigos, puesto que $f(n) < c\,g(n) < d\,g(n)$ siempre que $n > M > N$.

Un método útil para encontrar un par de testigos es seleccionar primero un valor N para el cual se pueda estimar el tamaño de $f(n)$ cuando $n > N$ y ver si podemos usar esta estimación para obtener un valor de c para el cual $f(n) < c\,g(n)$ siempre que $n > N$. Este método se ilustra en el siguiente ejemplo.

Ejemplo 2.7. Demuestra que $f(n) = n^2 + 2n + 1$ es $O(n^2)$.

Solución: Observamos que podemos estimar el tamaño de $f(n)$ cuando $n > 1$ porque $n < n^2$ y $1 < n^2$ para $n > 1$. Por tanto, se sigue que

$$0 \leq n^2 + 2n + 1 < n^2 + 2n^2 + n^2 = 4n^2$$

siempre que $n > 1$, como se muestra en la Figuara 2.3. Por consiguiente, podemos tomar $c = 4$ y $N = 1$ como testigos para mostrar que $f(n)$ es $O(n^2)$. Esto es, $f(n) = n^2 + 2n + 1 < 4n^2$ siempre que $n > 1$.

Alternativamente, podemos estimar el tamaño de $f(n)$ cuando $n > 2$. Si $n > 2$, tenemos que $2n < n^2$ y $1 < n^2$. Por tanto, si $n > 2$, tenemos

$$0 \leq n^2 + 2n + 1 < n^2 + n^2 + n^2 = 3n^2.$$

Se sigue de aquí $\tilde{c} = 3$ y $\tilde{N} = 2$ son también testigos de la relación $f(n) = O(n^2)$.

Observa que en la relación $f(n) = O(n^2)$, n^2 se puede reemplazar por cualquier función que tome valores mayores que n^2. Por ejemplo, $f(n) = O(n^3)$ o $f(n) = O(n^2 + n + 7)$.

También se cumple que n^2 es $O(n^2 + 2n + 1)$, porque $n^2 < n^2 + 2n + 1$ si $n > 1$. Esto significa que $c' = 1$ y $N' = 1$ son testigos de la relación $n^2 = O(n^2 + 2n + 1)$.

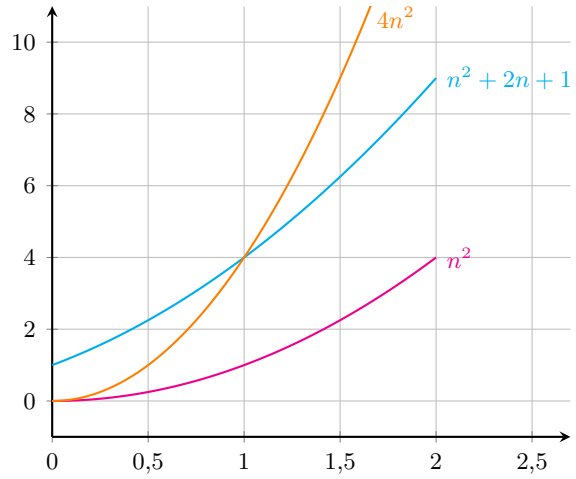


Figura 2.3: $n^2 + 2n + 1 = O(n^2)$: $n^2 + 2n + 1 < 4n^2$ para $n > 1$.

Observa que en el Ejemplo 2.7 tenemos dos funciones, $f(n) = n^2 + 2n + 1$ y $g(n) = n^2$, tales que $f(n) = O(g(n))$ y $g(n) = O(f(n))$ – lo último se deduce de la desigualdad $n^2 < n^2 + 2n + 1$, que se cumple para todo número entero positivo n. Decimos que las dos funciones $f(n)$ y $g(n)$ que satisfacen estas dos relaciones O son del **mismo orden**. Volveremos a esta notación en esta sección.

Observación 2.1. El hecho de que $f(n)$ sea $O(g(n))$ se escribe a veces como $f(n) = O(g(n))$. No obstante, el signo de igualdad en esta notación *no* representa genuinamente una igualdad. Al contrario, esta notación nos dice que se mantiene una desigualdad que relaciona los valores de las funciones f y g para números suficientemente grandes del dominio de ambas.

Cuando $f(n) = O(g(n))$ y $h(n)$ es una función positiva mayor que $g(n)$ para valores suficientemente grandes de n, se verifica que $f(n) = O(h(n))$. En otras palabras, la función $g(n)$ en la relación $f(n) = O(g(n))$ se puede remplazar por una función mayor. Para ver esto, observa que si

$$f(n) < c\,g(n) \quad \text{si} \quad n > N,$$

y si $h(n) > g(n)$ para todo $n > N$, entonces

$$f(n) < c\,h(n) \quad \text{si} \quad n > N;$$

por tanto, $f(n) = O(h(n))$.

Cuando se utiliza la notación O, la función g en la relación $f(n) = O(g(n))$ se elige tan pequeña como sea posible (a veces de un conjunto de funciones de referencia, como las funciones de la forma n^k, donde k es un entero positivo).

Los siguientes ejemplos ilustran cómo se usa la notación O para estimar el crecimiento de funciones.

Ejemplo 2.8. Demostrar que $7n^2 = O(n^3)$.

Solución: Observa que cuando $n > 7$, tenemos que $7n^2 < n^3$. Podemos obtener esta desigualdad multiplicando ambos lados de $7 < n$ por n^2. Por tanto, podemos tomar $c = 1$ y $N = 7$ como testigos para

establecer la relación $7n^2 = O(n^3)$. De forma alternativa, cuando $n > 1$, tenemos que $7n^2 < 7n^3$, por lo que $c' = 7$ y $N' = 1$ son también testigos de la relación $7n^2 = O(n^3)$.

Ejemplo 2.9. El ejemplo anterior muestra que $7n^2 = O(n^3)$. ¿Es cierto también que $n^3 = O(7n^2)$?

Solución: Para determinar si $n^3 = O(7n^2)$ necesitamos saber si hay dos constantes c y N tales que $n^3 < c(7n^2)$ siempre que $n > N$. La desigualdad $n^3 < c(7n^2)$ es equivalente a $n < 7c$, que viene de dividir la desigualdad original por la conatidad positiva n^2. Observa que no existe c alguno para el cual $n < 7c$ para todo $n > N$, no importa el valor de N, porque n se puede hacer arbitrariamente grande. Por tanto, no existen testigos c y N para la relación O propuesta. Así, n^3 *no es* $O(7n^2)$.

2.6.2. Algunos resultados importantes sobre la notación O

Los polinomios se utilizan a menudo para estimar el crecimiento de las funciones. En lugar de analizar el crecimiento de los polinomios cada vez que se presentan, nos vendría bien disponer de un resultado que se pudiese emplear genéricamente para estimar el crecimiento de un polinomio. El siguiente teorema proporciona tal resultado. Muestra que el término de mayor grado del polinomio domina su crecimiento, afirmando que un polinomio de grado k o inferior es $O(n^k)$.

Teorema 2.6.2. *Sea* $f(n) = a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0$, *donde* $a_0, a_1, \ldots, a_{k-1}, a_k$ *son números enteros no negativos. Entonces,* $f(n) = O(n^k)$.

Demostración. Si $n > 1$,

$$\begin{aligned} f(n) &= a_k n^k + a_{k-1}n^{k-1} + \cdots + a_1 n + a_0 \\ &= n^k \left(a_k + \frac{a_{k-1}}{n} + \cdots + \frac{a_1}{n^{k-1}} + \frac{a_0}{n^k}\right) \\ &< n^k(a_k + a_{k-1} + \cdots + a_1 + a_0). \end{aligned}$$

Por lo que

$$f(n) < c\,n^k,$$

donde $c = a_k + a_{k-1} + \cdots + a_1 + a_0$ siempre que $n > 1$. Por lo tanto, los testigos $c = a_k + a_{k-1} + \cdots + a_1 + a_0$ y $N = 1$ muestran que $f(n) = O(n^k)$. ■

Ejemplo 2.10. ¿Cómo se puede utilizar la notación O para estimar la suma de los primeros n enteros positivos?

Solución: Como cada uno de los números en la suma de los primeros n enteros no es mayor que n, tenemos

$$1 + 2 + \cdots + n < n + n + \cdots + n = n^2.$$

De esta desigualdad, se sigue que $1 + 2 + 3 + \cdots + n$ es $O(n^2)$, tomando $c = 1$ y $N = 1$ como testigos.

En el siguiente ejemplo se harán estimaciones en notación O para la función factorial y su logaritmo. Estas estimaciones serán importantes en el análisis de la complejidad de los procedimientos de ordenación.

Ejemplo 2.11. Da una estimación con la notación O para la función factorial y su logaritmo. La función factorial $f(n) = n!$ se define como

$$n! = 1 \cdot 2 \cdot 3 \cdots n$$

para cada entero positivo n y $0! = 1$.

Solución: Se puede obtener una estimación de $n!$ en notación O teniendo en cuenta que cada término del producto no es mayor que n. Por tanto,

$$n! = 1 \cdot 2 \cdot 3 \cdots n < n \cdot n \cdot n \cdots n = n^n.$$

Esta desigualdad muestra que $n! = O(n^n)$, tomando $c = 1$ y $N = 1$ como testigos. Tomando logaritmos en ambos miembros de la desigualdad establecida para $n!$, obtenemos

$$\log n! < \log n^n = n \log n.$$

Esto implica que $\log n! = O(n \log n)$, de nuevo considerando $c = 1$ y $N = 1$ como testigos.

Ejemplo 2.12. Más adelante demostraremos que

$$n < 2^n$$

para todo entero positivo n. Usando esta desigualdad, se puede concluir que $n = O(2^n)$. (Toma $c = N = 1$ como testigos.) Como la función logaritmo es creciente, tomando logaritmos (en base 2) en ambos lados de la desigualdad, se ve que

$$\log n < n.$$

Por tanto,

$$\log n = O(n)$$

(de nuevo tomamos $c = N = 1$ como testigos).

Si tenemos logaritmos en base b, $b \neq 2$, seguimos teniendo que $\log_b n = O(n)$, puesto que

$$\log_b n = \frac{\log n}{\log b} < \frac{n}{\log b}$$

siempre que n sea un entero positivo. Tomamos $c = 1/\log b$ y $N = 1$ como testigos.

Como se mencionó anteriormente, la notación O se usa para estimar el número de operaciones que se realizan para resolver un problema utilizando un determinado procedimiento o algoritmo. Si denotamos por $f(n) < g(n)$ el hecho de que $f(n) = O(g(n))$, entonces podemos ordenar de menor a mayor el comportamiento asintótico de algunas funciones que aparecen con más frecuencia en esta asignatura:

$$\boxed{1 < \log(n) < n < n\log(n) < n^2 < n^3 < \cdots < 2^n < n! < n^n.}$$

Haciendo uso de propiedades del cálculo infinitesimal, se puede ver que el cociente entre cada una de estas funciones y la función que le sigue tiende a cero cuando n crece indefinidamente. La Figura 2.4 presenta las gráficas de estas funciones, empleando en cada una de ellas una escala logarítmica.

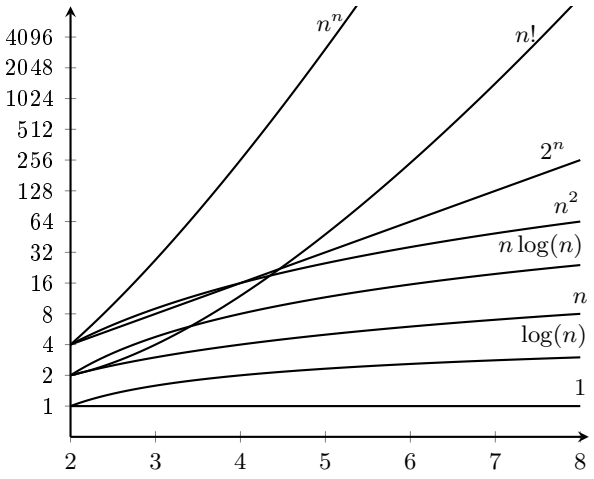


Figura 2.4: Comparación del crecimiento de funciones utilizadas comúnmente en las estimaciones con la notación O.

2.6.3. Crecimiento de combiandción de funciones

Muchos algoritmos consisten en dos o más subprocesos separados. El número de pasos realizados por un ordenador para solucionar un problema con una entrada de tamaño dado usando tales algoritmos es la suma del número de pasos realizados en cada uno de los subprocesos. Para dar una estimación en notación O del número total de pasos requeridos, se necesita obtener estimaciones en notación O para el número de pasos en cada uno de los subprocesos y combinar estas estimaciones.

Si se es cuidadoso, se puede dar una estimación en notación O de la combinación de varias funciones teniendo en cuenta las diferentes estimaciones O dadas para cada una de ellas. En particular, a menudo es necesario estimar el crecimiento de la suma y el producto de dos funciones. ¿Qué puede decirse si se conocen estimaciones en notación O para cada una de las dos funciones? Para ver qué clase de estimación se tiene para la suma y el producto de dos funciones, supongamos que $f_1(n) = O(g_1(n))$ y $f_2(n) = O(g_2(n))$.

De la definición de la notación O, se sigue que hay cuatro constantes c_1, c_2, N_1 y N_2 tales que

$$f_1(n) < c_1\, g_1(n)$$

cuando $n > N_1$ y

$$f_2(n) < c_2\, g_2(n)$$

cuando $n > N_2$. Para estimar la suma de $f_1(n)$ y $f_2(n)$, observa que, cuando n es mayor que N_1 y N_2 se deduce de las desigualdades para $f_1(n)$ y $f_2(n)$ que

$$f_1(n) + f_2(n) < c_1\, g_1(n) + c_2\, g_2(n) < c_1\, g(n) + c_2\, g(n) = (c_1 + c_2)\, g(n) = c\, g(n),$$

donde $c = c_1 + c_2$ y $g(n) = \text{máx}\{g_1(n), g_2(n)\}$. (Aquí, $\text{máx}\{a, b\}$) denota el máximo, o el mayor, de a y b.)

Esta desigualdad muestra que $f_1(n) + f_2(n) < c\, g(n)$ siempre que $n > N$, donde $N = \text{máx}\{N_1, N_2\}$. Enunciamos este resultado en el siguiente teorema.

Teorema 2.6.3. *Supongamos que* $f_1(n) = O(g_1(n))$ *y* $f_2(n) = O(g_2(n))$. *Entonces* $f_1(n) + f_2(n) = O(\text{máx}\{g_1(n), g_2(n)\})$.

A menudo tenemos estimaciones en notación O para las funciones f_1 y f_2 en términos de una misma función g. En esta situación, se puede usar el teorema anterior para mostrar que $f_1(n) + f_2(n)$ es también $O(g(n))$, puesto que máx$\{g(n), g(n)\} = g(n)$. Este resultado se enuncia en el siguiente corolario.

Corolario 2.6.4. *Supongamos que $f_1(n)$ y $f_2(n)$ son ambas $O(g(n))$. Entonces $f_1(n) + f_2(n) = O(g(n))$.*

De forma similar, se pueden derivar estimaciones en notación O para el producto de las funciones f_1 y f_2. Cuando n es mayor que máx$\{N_1, N_2\}$, se cumple que

$$f_1(n)\, f_2(n) < (c_1\, g_1(n))\, (c_2\, g_2(n)) = (c_1\, c_2)\, (g_1(n)\, g_2(n)) = c\, g_1(n)\, g_2(n),$$

donde $c = c_1\, c_2$. De esta desigualdad, se sigue que $f_1(n)\, f_2(n) = O(g_1(n)\, g_2(n))$, puesto que hay dos constantes c y N, que son $c = c_1\, c_2$ y $N =$ máx$\{N_1, N_2\}$, tales que $f_1(n)\, f_2(n) < c\, (g_1(n)\, g_2(n))$ siempre que $n > N$. Se enuncia este resultado en el siguiente teorema.

Teorema 2.6.5. *Supongamos que $f_1(n) = O(g_1(n))$ y $f_2(n) = O(g_2(n))$. Entonces tenemos que $f_1(n) f_2(n) = O(g_1(n) g_2(n))$.*

El objetivo de utilizar la notación O en la estimación de funciones es elegir una función $g(n)$ que crezca con la relativa lentitud de tal forma que $f(n)$ sea $O(g(n))$. Los ejemplos siguientes ilustran cómo usar los dos teoremas anteriores para hacer esto. El tipo de análisis que se lleva a cabo en estos ejemplos se emplea frecuentemente en el cómputo del tiempo requerido para resolver problemas utilizando programas de ordenador.

Ejemplo 2.13. Da una estimación en notación O para $f(n) = 3n \log(n!) + (n^2 + 3) \log n$, donde n es un entero positivo.

Solución: Primero estimaremos el producto $3n \log(n!)$. Del Ejemplo 2.11 se sabe que $\log(n!)$ es $O(n \log n)$. Utilizando esta estimación y el hecho de que $3\, n = O(n)$, el Teorema 2.6.5 afirma que la estimación para $3n\, log(n!)$ es $O(n^2 \log n)$.

Ahora se estima el producto $(n^2 + 3) \log n$. Como $n^2 + 3 < 2\, n^2$ cuando $n > 2$, se sigue que $n^2 + 3$ es $O(n^2)$. Por tanto, por el Teorema 2.6.5, $(n^2 + 3) \log n = O(n^2 \log n)$. Usando el Corolario 2.6.4 para combinar las dos estimaciones dadas, se ve que $f(n) = 3n \log(n!) + (n^2 + 3) \log n = O(n^2 \log n)$.

Ejemplo 2.14. Da una estimación en notación O para $f(n) = (n + 1) \log(n^2 + 1) + 3n^2$.

Solución: Primero, daremos una estimación para $(n+1) \log(n^2+1)$. Observa que $n+1 = O(n)$. Además, $n^2 + 1 < 2n^2$ cuando $n > 1$. Por tanto,

$$\log(n^2 + 1) < \log(2n^2) = \log 2 + \log n^2 = \log 2 + 2 \log n < 3 \log n$$

si $n > 2$. Esto muestra que $\log(n^2 + 1) = O(\log n)$.

Del Teorema 2.6.5 se sigue que $(n + 1) \log(n^2 + 1) = O(n \log n)$. Como $3n^2 = O(n^2)$, el Teorema 2.6.3 dice que $f(n) = O(\text{máx}\{n \log n, n^2\})$. Como $n \log n < n^2$ para $n > 1$, se sigue que $f(n) = O(n^2)$.

2.6.4. Las notaciones Omega y Zeta

La notación O se utiliza frecuentemente para describir el crecimiento de funciones, pero tiene algunas limitaciones. En particular, cuando $f(n) = O(g(n))$, tenemos una cota superior en términos de $g(n)$ del tamaño de $f(n)$ para valores grandes de n. Sin embargo, la notación O no nos proporciona una cota inferior del tamaño de $f(n)$ para valores grandes de n. Para esto utilizaremos la **notación Omega**. Cuando queremos dar tanto una cota inferior como una cota superior del tamaño de la función $f(n)$ en términos de una función de referencia $g(n)$ usamos la **notación Zeta**.

Definimos a continuación la notación Omega e ilustramos su uso. Tras ella, damos la definición de la notación Zeta.

Definición 2.6.6. *Sean $f, g : \mathbb{N} \to \mathbb{N}$ funciones positivas. Diremos que $f(n) = \Omega(g(n))$ si y solo si*

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} > 0.$$

Equivalentemente, $f(n) = \Omega(g(n))$ si y solo si para cada entero $N > 0$ existe una constante $k > 0$ tal que

$$f(n) > k\,g(n), \quad n > N.$$

Esta expresión se lee "$f(n)$ es omega de $g(n)$" o bien "$f(n)$ es una omega mayúscula de $g(n)$".

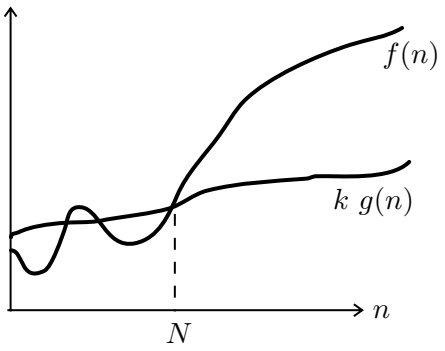


Figura 2.5: $f(n) = \Omega(g(n))$.

Ejemplo 2.15. La función $f(n) = 8n^3 + 5n^2 + 7$ es $\Omega(g(n))$, donde $g(n) = n^3$. Esto es fácil de ver, ya que $f(n) = 8n^3 + 5n^2 + 7 > 8n^3$ para todo $n > 1$. Esto es equivalente a decir que $g(n) = n^3$ es $O(8n^3 + 5n^2 + 7)$, lo cual se establece directamente sin más que cambiar el sentido de la desigualdad.

Es importante que, por motivación propia, el estudiante relacione los conceptos estudiados. Por ejemplo, es posible relacionar las notaciones O y *Omega* de la siguiente manera: observemos que si $f(n) = O(g(n))$, entonces para cada entero $N > 0$ existe una constante $c > 0$ tal que

$$f(n) < c\,g(n), \quad n > N.$$

Pero como $c > 0$, entonces podemos dividir ambos lados de la desigualdad por c, obteniendo

$$\frac{1}{c} f(n) < g(n), \quad n > N.$$

Esto significa que para cada entero $N > 0$ existe una constante $k = c^{-1} > 0$ tal que se cumple la desigualdad anterior. Por la definición de $\Omega(\cdot)$, hemos descubierto que que si $f(n) = O(g(n))$, entonces $g(n) = \Omega(f(n))$.

También es posible empezar desde cero y argumentar en el sentido opuesto, es decir, podemos suponer que $g(n) = \Omega(f(n))$ y deducir que $f(n) = O(g(n))$ (aquí nos olvidaremos de que $k = c^{-1}$ y buscaremos nuevamente la relación entre estas dos constantes). En efecto, si $g(n) = \Omega(f(n))$ entonces para cada entero $N > 0$ existe una constante $k > 0$ tal que

$$k\, f(n) < g(n), \quad n > N.$$

Pero como $k > 0$, entonces podemos dividir ambos lados de la desigualdad por k, obteniendo

$$f(n) < \frac{1}{k}\, g(n), \quad n > N.$$

Esto significa que para cada entero $N > 0$ existe una constante $c = k^{-1} > 0$ tal que se cumple la desigualdad anterior. Por la definición de $O(\cdot)$, hemos descubierto que que si $g(n) = \Omega(f(n))$, entonces $f(n) = O(g(n))$.

En conclusión, hemos descubierto la siguiente relación entre $O(\cdot)$ y $\Omega(\cdot)$:

$$\boxed{f(n) = O(g(n)) \text{ si y solo si } g(n) = \Omega(f(n)).}$$

A menudo es importante conocer el orden de crecimiento de una función en términos de una función de referencia relativamente simple, como puede ser n^k donde k es un racional positivo, o c^n, para $c > 1$. Saber el orden de crecimiento requiere que tengamos tanto una cota superior como una cota inferior para el tamaño de la función. Esto es, dada una función $f(n)$, buscamos una función de referencia $g(n)$ tal que $f(n)$ sea $O(g(n))$ y $\Omega(g(n))$. La notación Zeta, definida a continuación, se utiliza para expresar estas dos relaciones, proporcionando una cota superior y otra inferior del tamaño de la función.

Definición 2.6.7. *Sean $f, g : \mathbb{N} \to \mathbb{N}$. Diremos que $f(n) = \Theta(g(n))$ si y solo si $f(n) = O(g(n))$ y $f(n) = \Omega(g(n))$. Es decir,*

$$0 < \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty.$$

Equivalentemente, $f(n) = \Theta(g(n))$ si y solo si para cada entero $N > 0$ existen constantes $c, k > 0$ tal que

$$k\, g(n) < f(n) < c\, g(n), \quad n > N.$$

Cuando $f(n) = \Theta(g(n))$, decimos que "f es zeta de g" o que "f es del orden de g".

Cuando $f(n) = \Theta(g(n))$, también se cumple que $g(n) = \Theta(g(n))$. Observa también que $f(n) = \Theta(g(n))$ si, y sólo si, $f(n) = O(g(n))$ y $g(n) = O(f(n))$.

Generalmente, cuando se usa la notación Zeta, la función $g(n)$ en $\Theta(g(n))$ es una función de referencia relativamente simple, tales como $n^k, c^n, \log n, \ldots$, mientras que $f(n)$ puede ser relativamente compleja.

Ejemplo 2.16. Vimos en el Ejemplo 2.10 que la suma de los n primeros enteros positivos era $O(n^2)$. ¿Es del orden de n^2 esta suma?

Solución: Sea $f(n) = 1 + 2 + 3 + \cdots + n$. Como ya sabemos que $f(n) = O(n^2)$, para mostrar que $f(n)$ es del orden de n^2 necesitamos encontrar una constante positiva c tal que $f(n) > c\, n^2$ para enteros positivos

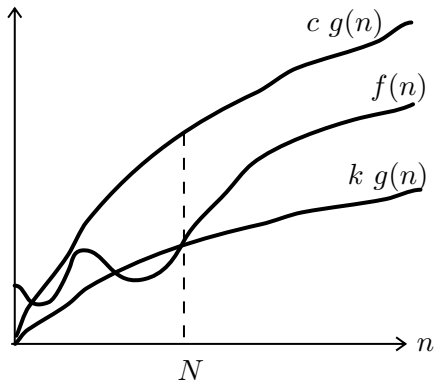


Figura 2.6: $f(n) = \Theta(g(n))$.

n suficientemente grandes. Para obtener una cota inferior de la suma, podemos ignorar la primera mitad de los términos. Sumando sólo los términos mayores que $\lceil n/2 \rceil$, tenemos

$$\begin{aligned} 1 + 2 + \cdots + n &\geq \lceil n/2 \rceil + (\lceil n/2 \rceil + 1) + \cdots + n \\ &> \lceil n/2 \rceil + \lceil n/2 \rceil + \cdots \lceil n/2 \rceil \\ &= (n - \lceil n/2 \rceil + 1)\lceil n/2 \rceil \\ &\geq (n/2)\,(n/2) \\ &= n^2/4. \end{aligned}$$

Esto muestra que $f(n) = \Omega(n^2)$. Concluimos que $f(n)$ es del orden de n^2, o utilizando la notación Θ, que $f(n) = \Theta(n^2)$.

Se puede ver que $f(n) = \Theta(g(n))$ si podemos encontrar dos números reales c_1 y c_2 y un número real postivo N tales que

$$c_1\, g(n) < f(n) < c_2\, g(n)$$

siempre que $n > N$. Esto muestra que $f(n) = O(g(n))$ y $f(n) = \Omega(g(n))$.

Ejemplo 2.17. Demuestra que $3n^2 + 8n \log n$ es $\Theta(n^2)$.

Solución: Como $0 \leq 8n \log n < 8n^2$, se sigue que $3n^2 + 8n \log n < (3 + 8)\, n^2 = 11n^2$ para $n > 1$. Por lo tanto, $3n^2 + 8n \log n$ es $O(n^2)$. Claramente, n^2 es $O(3n^2 + 8n \log n)$. Así, $3n^2 + 8n \log n$ es $\Theta(n^2)$.

Un hecho importante es que el término de mayor grado del polinomio determina su orden. Por ejemplo, si $f(n) = 3n^5 + n^4 + 17n^3 + 2$, entonces $f(n)$ es del orden de n^5. Esto se enuncia en el teorema siguiente, cuya demostración se propone como ejercicio.

Teorema 2.6.8. *Sea $f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$, donde $a_0, a_1, \ldots, a_k$ son números enteros positivos con $a_k \neq 0$. Entonces, $f(n)$ es del orden de n^k.*

Ejemplo 2.18. Los polinomios $3n^8 + 10n^7 + 221n^2 + 2$, $n^{19} + 18n^4 + 10$ y $n^{99} + 40n^{98} + 100n$ son del orden n^8, n^{19} y n^{99}, respectivamente.

2.7. Complejidad de algoritmos

¿Cuándo proporciona un algoritmo una solución satisfactoria a un problema? Primero, debe producir siempre la respuesta correcta. En segundo lugar, debería ser eficiente. En esta sección estudiaremos la eficiencia de los algoritmos.

¿Cómo se puede analizar la eficiencia de los algoritmos? Una medida de eficiencia es el tiempo que requiere un ordenador para resolver un problema utilizando un algoritmo para valores de entrada de un tamaño especificado. Una segunda medida es la cantidad de memoria que se necesita de nuevo para valores de entrada de un tamañao dado.

Preguntas como las anteriores están ligadas a la **complejidad computacional** del algoritmo. Un análisis del tiempo requerido para resolver un problema de un tamaño particular está relacionado con la **complejidad en tiempo** del algoritmo. Análogamente, un análisis de la memoria de ordenador requerida involucra la **complejidad en espacio** del algoritmo. Las consideraciones de complejidad en tiempo y en espacio de un algoritmo son esenciales en al implementación de algoritmos. Obviamente, es importante saber si un algoritmo producirá su respuesta en un milisegundo, en un minuto o en un millón de años. De forma similar, debemos tener suficiente memoria disponible para poder resolver el problema, por lo que la complejidad en espacio debe tenerse también en cuenta.

Las consideraciones sobre complejidad en espacio están ligadas a las estructuras de datos usadas en la implementación del algoritmo. Como las estructuras de datos no se tratan con detalle en esta asignatura, no consideraremos la complejidad en espacio. Restringiremos nuestra atención a la complejidad en tiempo, de manera que, a menos que se diga lo contrario, cuando hablamos de complejidad nos estaremos refiriendo a la complejidad en tiempo.

2.7.1. Complejidad en tiempo

La complejidad en tiempo de un algoritmo se puede expresar en términos del número de operaciones que realiza el algoritmo cuando los datos de entrada tienen un tamaño particular. Las operaciones utilizadas para medir la complejidad pueden ser comparaciones de enteros, la suma, multiplicación o división de enteros, así como cualquier otra operación básica.

La complejidad se describe en términos del número de operaciones requeridas, en lugar del tiempo de cálculo real, debido a que distintos ordenadores necesitan tiempos diferentes para realizar las mismas operaciones básicas. Los ordenadores más rápidos pueden realizan operaciones básicas con bits (por ejemplo, sumar, multiplicar, comparar o intercambiar las posiciones de los bits) en 10^{-9} segundos (un nanosegundo), mientras que un ordenadore personal puede requerir 10^{-6} segundos (un microsegundo), mil veces más tiempo, para realizar la misma operación. Además, es bastante complicado descomponer todas las operaciones que desarrolla un algoritmo en las operaciones elementales que realiza un ordenador.

Ilustramos a continuación la complejidad del algoritmo MAX de la Sección 2.2, que busca el elemento máximo de un conjunto.

Ejemplo 2.19. Describe la complejidad del algoritmo MAX de la Sección 2.2 para encontrar el elemento máximo de un conjunto.

Solución: Se usará como medida de la complejidad en tiempo del algoritmo el número de comparaciones, pues la comparación es la operación básica utilizada.

Para encontrar el máximo elemento de un conjunto con n elementos, listados en orden arbitrario, primero se fija el máxima preliminar como el elemento inicial de la lista. Luego, tras realizar una comparación para determinar si se ha llegado al final de la lista, se comparan el máximo preliminar y el segundo elemento de la lista, actualizando el valor del máximo si el segundo término es mayor que el primero. Este procedimiento se repite utilizando dos comparaciones adicionales por cada término: una para determinar si se ha llegado al fin de la lista y otra para determinar si procede utilizar el valor provisional del máximo. Como se hacen dos comparaciones desde el segundo hasta el n-ésimo elemento, y una más para salir del bucle cuando $i = n+1$, se realizan exactamente $2(n-1)+1 = 2n-1$ comparaciones al aplicar el algoritmo. Por tanto, el algoritmo para encontrar el máximo de un conjunto de n elementos tiene una complejidad $\Theta(n)$, medido en términos del número de comparaciones realizadas.

Ahora analizaremos la complejidad en tiempo de los algoritmos de búsqueda.

Ejemplo 2.20. Calcula la complejidad en tiempo del algoritmo de búsqueda lineal.

Solución: Para medir la complejidad en tiempo utilizaremos de nuevo el número de comparaciones realizadas por el algoritmo. En cada paso del bucle del algoritmo se llevan a cabo dos comparaciones: una para ver si se ha alcanzado el final de la lista y la segunda para comparar el elemento x con un término de la lista. Finalmente, fuera del bucle se hace una comparación más. Por tanto, si $x = a$, se hacen $2i+1$ comparaciones. El mayor número de comparaciones se alcanza cuando el elemento no está en la lista, a saber, $2n+1$. En este caso, se hacen $2n$ comparaciones para determinar que $x \neq a$, para $i = 1, 2, \ldots, n$, una comparación adicional para salir del bucle y una más fuera del bucle. Por tanto, cuando x no figura en la lista, se hace un total de $2n+2$ comparaciones. Así, una búsqueda lineal requiere a lo más $O(n)$ comparaciones.

2.7.2. Complejidad del peor caso

El tipo de análisis de complejidad en el Ejemplo 2.20 es un análisis de **peor caso**. Por comportamiento de un algoritmo en el peor caso entendemos el mayor número de operaciones que hace falta para resolver el problema dado utilizando el algoritmo para unos datos de entrada de un determinado tamaño. Los análisis del peor caso nos dicen cuántas operaciones tienen que realizar los algoritmos para garantizar que producirán una solución.

Ejemplo 2.21. Calcula la complejidad del algoritmo de búsqueda binaria.

Solución: Por simplicidad, suponemos que hay $n = 2^k$ elementos en la lista $a_1, a_2, \ldots, a_n$, donde k es un entero no negativo. Observa que $k = \log n$. (Si n, el número de elementos de la lista, no es una potencia de 2, la lista se puede considerar parte de otra más larga con 2^{k+1} elementos, donde $2^k < n < 2^{k+1}$. Aquí, 2^{k+1} es la menor potencia de 2 mayor que n.)

En cada paso del algoritmo, i y j, las posiciones del primero y último término de la lista restringida considerada en este paso se comparan para ver si la lista restringida tiene más de un término. Si $i < j$, se realiza una comparación para determinar si x es mayor que el término centrar de la lista restringida.

En el primer paso, la búsqueda se restringe a una lista con 2^{k-1} términos. Hasta aquí se han hecho dos comparaciones. Este procedimiento continúa realizando dos comparaciones en cada paso para restringir la

búsqueda a una lista con la mitad de términos. En otras palabras, se hacen dos comparaciones en el primer paso del algoritmo cuando la lista tiene 2^k elementos, dos más cuando la búsqueda se ha reducido a una lista con 2^{k-1} elementos, dos más cuando la búsqueda se reduce a 2^{k-2} elementos y así sucesivamente. Las dos últimas comparaciones se hacen cuando la búsqueda se ha reducido a una lista con 2^1 elementos. Finalmente, cuando sólo queda un término en la lista, una comparación basta para saber que no hay términos adicionales, y se necesita una más para determinar si este término es x.

Por tanto, a lo más, se requiere $2k + 2 = 2\log n + 2$ comparaciones para realizar una búsqueda binaria cuando la lista de entrada tiene 2^k elementos. (Si n no es potencia de 2, la lista original se amplia a una con 2^{k+1} términos, donde $k = \lfloor \log n \rfloor$, y la búsqueda requiere a lo más $2\lceil \log n \rceil + 2$). Consecuentemente, una búsqueda binaria requiere como máximo $\Theta(\log n)$ comparaciones. De este análisis se concluye que el algoritmo de búsqueda binaria es más eficiente, en el peor caso, que la búsqueda lineal.

2.7.3. Complejidad del caso promedio

Otro tipo importante de análisis de complejidad, además del peor caso, es el denominado análisis del **caso promedio**. En este tipo de análisis se busca el número promedio de operaciones realizadas para solucionar un problema considerando todas las posibles entradas de un tamaño determinado. En análisis de complejidad del caso promedio es, generalmente, mucho más complicado que el análisis del peor caso. No obstante, podemos hacer este análisis para el algoritmo de búsqueda lineal sin gran dificultad, como se muestra en el siguiente ejemplo.

Ejemplo 2.22. Describe el comportamiento en el caso promedio del algoritmo de la búsqueda lineal suponiendo que el elemento x está en la lista.

Solución: Hay n tipos de posibles entradas cuando sabemos que x está en la lista. Si x es el primer elemento de la lista, se necesitan tres comparaciones, una para determinar si se ha alcanzado el final de la lista, una para comparar x con el primer término y una fuera del bucle. Si x es el segundo término de la lista, se necesitan dos comparaciones más, de tal forma que se hacen un total de cinco comparaciones. En general, si x es el término i-ésimo de la lista, se realizarán dos comparaciones en cada uno de los i pasos del bucle, y uno fuera del bucle de tal forma que se necesita un total de $2i + 1$ comparaciones. Por tanto, el número promedio de comparaciones realizadas es igual a

$$\frac{3 + 5 + 7 + \cdots + (2n + 1)}{n} = \frac{2(1 + 2 + 3 + \cdots + n) + n}{n}.$$

De la asignatura **Matemáticas Discretas** sabemos que

$$1 + 2 + 3 + \cdots + n = \frac{n(n + 1)}{2}.$$

(Esta identidad se puede demostrar utilizando el Principio de Inducción Matemática.) Por tanto, el número promedio de comparaciones realizadas por el algoritmo de búsqueda lineal (cuando se sabe que x está en la lista) es

$$\frac{2[n(n + 1)/2]}{n} + 1 = n + 2,$$

que es $\Theta(n)$.

Observación 2.2. En el análisis del ejemplo se asume que x está en la lista que se busca y puede estar en cualquier posición. También se puede hacer un análisis del caso promedio de este algoritmo cuando es posible que x no esté en la lista.

Observación 2.3. Generalmente, las comparaciones que se necesitan para determinar si hemos llegado al final de un bucle no se tiene en cuenta. Aunque en los ejemplos anteriores las hemos considerado, a partir de ahora las ignoraremos.

2.7.4. Complejidad del peor caso de dos algoritmos de ordenación

Analizamos la complejidad de la ordenación por el método de la burbuja y la ordenación por inserción de la Sección 2.4.

Ejemplo 2.23. ¿Cuál es la complejidad en el peor caso de la ordenación por el método de la burbuja en términos del número de comparaciones realizadas?

Solución: La ordenación por el método de la burbuja ordena una lista desarrollando una secuencia de pasadas sobre la lista. Durante cada pasada, este método de ordenación compara sucesivamente elementos adyacentes, intercambiándolos cuando es necesario. Cuando comienza la pasada i-ésima, el método garantiza que los $i-1$ elementos mayores están en sus posiciones correctas. Durante esta pasada se hacen $n-i$ comparaciones. Por tanto, el número total de comparaciones realizadas por el método de la burbuja para ordenar una lista de n elementos es

$$(n-1)+(n-2)+\cdots+2+1 = \frac{(n-1)\,n}{2}$$

empleando la fórmula para la suma de los primeros $n-1$ enteros. Observa que la ordenación por el método de la burbuja hace siempre este número de comparaciones, porque continúa incluso si la lista está completamente ordenada en algunos pasos intermedios. Consecuentemente, este método realiza $(n-1)n/2$ comparaciones, por lo que tiene complejidad en el peor caso $\Theta(n^2)$ en términos del número de comparaciones realizadas.

Ejemplo 2.24. ¿Cuál es la complejidad en el peor caso de la ordenación por inserción en términos de números de comparaciones realizadas?

Solución: La ordenación por inserción coloca el elemento j-ésimo en la posición correcta entre los primeros $j-1$ términos que ya hemos puesto en el orden correcto. Esto se hace usando una técnica de búsqueda lineal, comparando el elemente j-ésimo con términos sucesivos hasta que se encuentre un elemento que sea mayor o igual que él, o se compara a_j consigo mismo y se para porque a_j no es menor que él mismo. Consecuentemente, en el peor caso, se requiere j comparaciones para insertar el elemento j-ésimo en la posición correcta. Así, el número total de comparaciones realizadas para ordenar por inserción una lista de n elementos es

$$2+3+\cdots+n = \frac{n\,(n+1)}{2}-1$$

empleando la fórmula de la suma de enteros consecutivos y teniendo en cuenta que el primer término, 1, no se considera en esta suma. Observa que el número de comparaciones realizadas en una ordenación por inserción depende considerablemente de la disposición de los elementos menores de la lista. Concluimos que la ordenación por inserción tiene una complejidad en el peor caso $\Theta(n^2)$.

Complejidad	Terminología
$O(1)$	Complejidad constante
$O(\log n)$	Complejidad logarítmica
$O(n)$	Complejidad lineal
$O(n \log n)$	Complejidad $n \log n$
$O(n^k)$	Complejidad polinómica
$O(b^n)$, donde $b > 1$	Complejidad exponencial
$O(n!)$	Complejidad factorial

Cuadro 2.1: Terminología comúnmente utilizada para la complejidad de algoritmos.

2.7.5. El significado de la complejidad de un algoritmo

La Tabla 2.7.5 muestra la terminología comúnmente utilizada para describir la complejidad de los algoritmos.

Un algoritmo tiene **complejidad polinómica** si tiene complejidad $O(n^k)$, donde k es un racional mayor o igual que 1. Por ejemplo, el algoritmo de ordenación por el método de la burbuja es una algoritmo de complejidad polinómica porque usa $O(n^2)$ comparaciones en el peor caso. Un algoritmo tiene **complejidad exponencial** si su complejidad es $O(b^n)$, donde $b > 1$. El algoritmo que determina si una fórmula proposicional con n variables puede ser satisfecha mediante la comprobación de todas las posibles asignaciones de valores de verdad es una algoritmo con complejidad exponencial porque utiliza $O(2^n)$ operaciones. Finalmente, un algoritmo tiene **complejidad factorial** si su complejidad es $O(n!)$. El algoritmo que genera todas las maneras en las que un viajante podría visitar n ciudades tiene complejidad factorial.

Un problema que se puede resolver utilizando un algoritmo con complejidad polinómica en el peor caso se llama **tratable**, pues se espera que el algoritmo produzca la solución al problema para una entrada de tamaño razonable en un tiempo relativamente corto. Sin embargo, si el polinomio de la estimación en notación O tiene un grado alto (como, or ejemplo, 100) o si los coeficientes son extremadamente grandes, el algoritmo puede necesitar un tiempo muy grande para solucionar el problema. Por tanto, el que ese problema se pueda solucionar utilizando un algoritmo con complejidad polinómica en el peor caso no es una garantía de que el problema se pueda solucionar en un tiempo razonable, incluso para valores de entrada pequeños. Afortunadamente, en la práctica, el grado y los coeficientes de los polinomios de tales estimaciones son pequeños.

La situación es mucho peor para problemas que no se pueden resolver utilizando un algoritmo con complejidad polinómica en el peor caso. Tales problemas se llaman **intratables**. Generalmente, pero no siempre, se requiere un tiempo extremadamente largo para resolver un problema en el peor caso incluso para un valor pequeño de entrada. En la práctica, no obstante, hay situaciones en las que un algoritmo puede resolver un problema mucho más rápidamente en la mayoría de los casos que en el peor caso. Cuando podamos permitir que algunos casos, quizá pocos, no se pueden resolver en un tiempo razonable, la complejidad en tiempo en el caso promedio es una medida más acertada del tiempo que llevaría un problema que el peor caso. Muchos de los problemas importantes en la industria son presentados como intratables, pero en la práctica pueden resolverse para, esencialmente, todos los conjuntos de datos que aparecen en la vida real.

Otra forma en la que se manejan los problemas intratables cuando aparecen en las aplicaciones prácticas es buscar soluciones aproximadas del problema en vez de soluciones exactas. Puede darse el caso de que existan algoritmos rápidos para encontrar soluciones aproximadas, quizá incluso con la garantía de que no difieran mucho de las soluciones exactas.

Existen algunos problemas para los cuales incluso se puede demostrar que no existen algoritmos que pueden resolverlos. Tales problemas se llaman **no resolubles** o **irresolubles** (en oposición a los problemas **resolubles**, para los cuales existen algoritmos que los resuelven).

El estudio de la complejidad de algortimos va mucho más allá de lo que podemos describir aquí. Ten en cuenta, no obstante, que se cree que muchos problemas resolubles tiene la propiedad de que no hay ningún algoritmo con complejidad polinómica en el peor caso que los resuelva, pero que en caso de conocerse alguna solución puede comprobarse que ésta, efectivamente, resuelve el problema en tiempo polinómico. Se dice que un problema pertenece a la **clase NP** si se puede comprobar en tiempo polinómico que una solución efectivamente lo es (los problemas tratables se dicen que pertenecen a la **clase P**)[1]. Hay también una clase importante de problemas, llamado **problemas NP-completos**, con la propiedad de que si alguno de estos problemas se puede resolver haciendo uso de un algoritmo con complejidad polinómica en el peor caso, entonces todos ellos se pueden resolver por medio de algoritmos con complejidad polinómica en el peor caso.

El problema de la satisfabilidad es un ejemplo de problema NP-completo – podemos verificar rápidamente que una asignación de valores de verdad a las variables de una fórmula proposicional la hace verdadera, pero no se ha descubierto un algoritmo con complejidad en tiempo polinómico que genere tal asignación de valores de verdad–. [Por ejemplo, una búsqueda exhaustiva sobre todos los posibles valores de verdad requiere $\Theta(2^n)$ operaciones con bits, donde n es el número de variables de la fórmula.] Además, si se conociese un algoritmo de complejidad polinómica para resolver este problema, podríamos encontrar algoritmos de complejidad polinómica para resolver todos los problemas de esta clase (y hay muchos problemas importantes en esta clase).

A pesar del gran esfuerzo invertido en investigación, no se han encontrado algoritmos con complejidad computacional polinómica en el peor caso para esta clase de problemas. Es generalmente aceptado, aunque no se ha demostrado, que ningún problema NP-completo puede resolverse en un tiempo polinómico.

Observa que una estimación en notación O para la complejidad en tiempo de un algoritmo expresa cómo el tiempo requerido para resolver un problema cambia a medida que el el tamaño de la entrada crece. En la práctica, se usa la mejor estimación que se pueda encontrar (esto es, la función de referencia más pequeña posible). No obstante, la estimación con la notación O de la complejidad no ofrece información sobre el tiempo de computación real utilizado. Una razón es que la estimación de la función O para la función $f(n)$ depende de dos constantes c y N ($f(n) = O(g(n))$ si $f(n) < c\,g(n)$ cuando $n > N$, donde c y N son constantes). Sin conocer las constantes c y N, la estimación no se puede utilizar para dara una cota superior del número de operaciones realizadas. Además, como se observó anteriormente, el tiempo requerido para completar una operación depende del tiempo de operación y del ordenador utilizado. (También se debe tener en cuenta que la notación O estima la complejidad del algoritmo porporcionando una cota superior, pero no inferior, del tiempo requerido en el peor caso para resolver un problema en función del tamaño de la entrada. Para proporcionar una cota inferior se debería utilizar la notación Zeta. No obstante, por simplicidad, utilizaremos la notación O para estimar la complejidad de algoritmos, aun sabiendo que la notación Zeta proporcionará

[1] NP proviene de la traducción inglesa de la expresión tiempo *polinómico no determinista*

Tamaño del problema	Operaciones con bits utilizadas					
n	$\log n$	n	$n \log n$	n^2	2^n	$n!$
10	3×10^{-9} s	10^{-8} s	3×10^{-8} s	10^{-7} s	10^{-6} s	3×10^{-3} s
10^2	7×10^{-9} s	10^{-7} s	7×10^{-7} s	10^{-5} s	4×10^{13} años	*
10^3	10^{-8} s	10^{-6} s	10^{-5} s	10^{-3} s	*	*
10^4	$1{,}3 \times 10^{-8}$ s	10^{-5} s	10^{-4} s	10^{-1} s	*	*
10^5	$1{,}7 \times 10^{-8}$ s	10^{-4} s	2×10^{-3} s	10 s	*	*
10^6	2×10^{-8} s	10^{-3} s	2×10^{-2} s	17 min	*	*

Cuadro 2.2: Tiempo de ordenador utilizado por los algoritmos.

siempre más información.)

Podemos determinar el tiempo real requerido por un algoritmo para solucionar un problema de tamaño especificado si todas las operaciones utilizadas se pueden reducir a las operaciones con bits que realiza el ordenador. La Tabla 2.7.5 muestra el tiempo necesario para resolver problemas de varios tamaños con un algoritmo que realiza el número de operaciones con bits que se indica. Con un asterisco se indican tiempos mañores a 10^{100} años. Para la construcción de esta tabla se ha considerado que cada operación con bits requiere 10^{-9} segundos. En el futuro, estos tiempos decrecerán con el aumento de velocidad de los ordenadores de nueva generación.

Es importante saber cuánto tiempo necesitará un ordenador para resolver un problema. Por ejemplo, si un algoritmo requiere 10 horas, puede valer la pena invertir el tiempo (y el dinero) requerido para resolver el problema. Pero si un algoritmo requiere diez mil millones de años para resolverlo, no sería razonable utilizar recursos para implementar este algoritmo. Uno de los fenómenos más interesantes de la tecnología moderna es el tremendo aumento de la memoria y la velocidad de los ordenadores. Otro factor importante que reduce el tiempo necesario para resolver los problemas con el ordenador es el **procesado en paralelo**, que es la técnica de realizar secuencias de operaciones simultáneamente.

Los algoritmos eficientes, incluidos muchos algoritmos con complejidad polinómica, son los que más se benefician de los avances tecnológicos significativos. Sin embargo, estos avances ayudan poco a superar la problemática de algoritmos con complejidad en tiempo exponencial o factorial. Debido al avance en la velocidad de cálculo, aumento de la memoria disponible y del uso del procesado en paralelo, muchos problemas que se consideraban imposibles de resolver hace cinco años ahora se resuelven de manera rutinaria, y de hecho, dentro de cinco años, esta afirmación volverá a ser cierta.

Capítulo 3

Introducción al pensamiento matemático

En los cursos de nivel superior como criptografía, el enfoque matemático evoluciona desde el cálculo hacia un énfasis en las definiciones, teoremas y demostraciones. Esta transición exige que los estudiantes interactúen con la matemática de maneras más abstractas y conceptuales. Este capítulo explora lo que se espera de los estudiantes en el ámbito universitario y proporciona estrategias para abordar y comprender las definiciones, los teoremas y las demostraciones. Aunque el enfoque principal se centra en el contenido de un curso de cálculo, las recomendaciones y estrategias presentadas son aplicables a una amplia variedad de cursos de matemáticas de nivel universitario (como la criptografía).

El Cálculo puede verse como el aparato matemático necesario para estudiar los número reales y las funciones definidas sobre ellos, comprender su estructura subyacente y aplicar lo aprendido en otros contextos. A diferencia de tu primer encuentro con la materia en la escuela secundaria, donde probablemente se enfatizaba el dominio de un recetario de procedimientos asociados a palabras claves en los enunciados de los problemas, nos enfocaremos en desarrollar la habilidad de enfrentarse a problemas nuevos no encontrados previamente en relaciones de ejercicios, deducir el método de resolución escogiendo las herramientas y pasos correctos sin que los indiquen explícitamente en el enunciado y a justificar de manera completa el planteamiento, desarrollo y conclusión de la soluciones. Estas espectativas se detallarán más adelante, así que no te preocupes si no sabes cómo alcanzarlas. El punto clave es que estás a punto de sumergirte en matemáticas serias, con énfasis en que alcances una comprensión profunda de las **definiciones**, **teoremas** y **demostraciones**.

Las matemáticas de nivel superior tienen mucho en común con las matemáticas de la escuela secundaria, y los estudiantes que han sido aceptados en la carrera de Ingeniería en Inteligencia Artificial ya poseen una serie de habilidades matemáticas que les serán útiles. Por otro lado, las matemáticas de nivel superior también difieren en algunos aspectos importantes. Esto significa que la mayoría de los estudiantes necesitan ampliar y adaptar sus habilidades existentes para continuar teniendo éxito. Hacer estas extensiones y adaptaciones puede ser difícil para aquellos que nunca han reflexionado realmente sobre la naturaleza de sus habilidades.

Algo que seguramente has aprendido a hacer es aplicar procedimientos matemáticos para calcular respuestas a preguntas estándar, tal vez combinando palabras clave en el enunciado de una pregunta con una receta memorizada de cálculos de un conjunto de muchas recetas. Esto podría haber sido útil para obtener una buena calificación en el examen de ingreso estandarizado a la universidad, que es un entorno controlado y predecible. Además, a algunas personas les gusta este tipo de trabajo. Les satisface llegar a una página de respuestas correctas y les gusta la seguridad de saber que, si hacen todo bien, sus respuestas serán correctas. A veces comparan favorablemente las matemáticas con otras materias en las que las cosas parecen ser más una cuestión de opinión y "no hay respuestas correctas".

A otras personas no les gusta este aspecto de las matemáticas. Les parece aburrido hacer muchos ejercicios repetitivos, y obtienen más satisfacción de aprender sobre *por qué* funcionan los diversos procedimientos y cómo encajan entre sí. Sin embargo, ten en cuenta que saber cómo aplicar los procedimientos es extremadamente importante porque, sin fluidez en los cálculos, es difícil concentrar tu atención en conceptos de nivel superior.

Cuando empieces a tomar cursos de nivel superior, los profesores esperarán que seas fluido en el uso de los procedimientos que ya has aprendido. Esperarán que seas capaz de manipular con precisión expresiones algebraicas, resolver ecuaciones, y así sucesivamente. Esperarán que puedas hacer estas cosas sin tener que detenerte cada vez a buscar una regla, y es posible que no tengan paciencia con los estudiantes que no pueden hacerlo. Esto no es porque sean impacientes con los estudiantes en general; la mayoría de los profesores estarán encantados de pasar mucho tiempo hablando contigo sobre nuevas matemáticas o respondiendo a estudiantes que digan: "Sé cómo hacer esto, pero nunca he entendido por qué lo hacemos de esta manera". Pero no esperarán volver a enseñar cosas que ya has estudiado. Por lo tanto, deberías repasar tu conocimiento antes de comenzar un curso, especialmente si, por ejemplo, no has hecho matemáticas en todo el verano.

Una vez que comiences, descubrirás que algunas matemáticas de nivel superior implican aprender nuevos procedimientos. Estos procedimientos, como era de esperar, serán más largos y complicados que los que encontraste en trabajos anteriores. Sin embargo, no nos preocupa tu capacidad para aplicar procedimientos largos y complicados, porque para haber llegado tan lejos debes ser capaz de hacer ese tipo de cosas. Aquí, queremos centrarnos en cambios más sustanciales en las formas en que debes interactuar con los procedimientos.

La primera diferencia sustancial es que tendrás más responsabilidad para decidir qué procedimiento aplicar. Por supuesto, ya has aprendido a hacer esto hasta cierto punto. Por ejemplo, has aprendido cómo expandir paréntesis y escribir cosas como:

$$(x+2)(x-5) = x^2 - 3x - 10.$$

Pero, con suerte, también has aprendido que *no* es sensato expandir cuando intentas simplificar una fracción como esta:

$$\frac{x^2(x+2)(x-5)}{x^2+2x}.$$

Para la fracción, la simplificación es más fácil si mantienes los factores "visibles". No obstante, muchos estudiantes automáticamente expanden, probablemente porque expandir fue una de las primeras cosas que aprendieron a hacer al estudiar álgebra. Serían, sin embargo, más exitosos en cursos de matemáticas como cálculo y álgebra lineal si aprendieran a detenerse y pensar primero en qué les permitiría avanzar más. Si esto no se aplica a ti para este tipo particular de problema, ¿se aplica en otros? ¿Alguna vez has hecho

un cálculo largo y luego te has dado cuenta de que no era necesario? ¿Podrías haberlo evitado si hubieras pensado primero? Parte de decidir qué procedimientos aplicar es darte un momento para pensarlo antes de lanzarte y hacer lo primero que te viene a la mente.

Esto podría no sonar como un gran problema, pero piensa por un momento en cuán a menudo *no* tienes que tomar una decisión sobre qué procedimiento aplicar. A menudo, las preguntas en los libros o en los exámenes te dicen exactamente qué hacer. Dicen cosas como: "Usa la regla de Cramer para resolver este sistema de ecuaciones". Incluso cuando una pregunta no te lo dice directamente, a veces es obvio por el contexto. En la escuela secundaria, si tu profesor dedicó una lección a mostrarte cómo aplicar el teorema de Rouché-Frobenius, luego te dio un conjunto de preguntas para hacer, probablemente era seguro asumir que éstas implicarían aplicar automáticamente el teorema de Rouché-Frobenius. Esto te ayudó, pero significa que gran parte del tiempo no tenías que decidir qué procedimiento aplicar. En el mundo en general, y en matemáticas avanzadas, tomar decisiones es más valorado y más esperado. Esto significa que las preguntas que se te presenten en hojas de problemas y exámenes generalmente solo dirán "resuelve este problema" en lugar de "resuelve este problema usando este procedimiento".

Otra parte de decidir qué procedimientos aplicar es ser capaz de distinguir entre casos que parecen similares pero que se abordan mejor de diferentes maneras. Por ejemplo, considera la integración, y más específicamente la integración por partes. Puede que sepas que esto se utiliza cuando queremos integrar un producto de dos funciones, una de las cuales se simplifica al derivarla, y la otra no se complica más al integrarla. Por ejemplo, en $\int xe^x dx$, x se simplifica si lo derivamos, y e^x no se complica más si lo integramos. Sin embargo, también puedes saber que a veces las situaciones matemáticas parecen superficialmente similares, pero se abordan mejor utilizando diferentes procedimientos. En el caso de la integración, la integración por sustitución podría ser más adecuada. Por ejemplo, en $\int xe^{x^2} dx$, probablemente querremos usar integración por sustitución en lugar de por partes. ¿Puedes ver por qué?

La integración por sustitución es un buen caso para otro punto que nos gustaría hacer, esta vez sobre tomar decisiones *dentro de* los procedimientos. Podría ser que leíste el final del último párrafo y pensaste: "Pero, ¿qué sustitución debería usar?" Tal vez tus profesores o libros siempre te dijeron qué usar, pero argumentaríamos que no necesariamente deberían tener que hacerlo. Después de un tiempo, deberías notar que ciertas sustituciones son útiles en ciertos casos. Si prestas atención a las estructuras de estos casos, entonces, incluso si no estarías seguro de poder elegir una buena sustitución para un caso nuevo, deberías tener una idea de algunas cosas sensatas para probar. Si no lo has pensado deliberadamente antes, te sugerimos que lo hagas ahora. Saca algunas preguntas sobre integración por sustitución y, sin hacer realmente los problemas, mira las sustituciones sugeridas. ¿Puedes anticipar por qué esas sustituciones funcionarán? ¿Puedes entonces anticipar qué funcionaría en casos similares?

Entonces, ¿cómo puede un estudiante mejorar su capacidad para tomar decisiones sobre y dentro de los procedimientos? Tenemos dos sugerencias. La primera es intentar hacer ejercicios de una fuente donde el procedimiento a aplicar no sea obvio. Un buen lugar para buscar es en los libros listados en la sección de bibliografía de la guía docente. La segunda sugerencia es convertir ejercicios ordinarios en oportunidades para la reflexión. Cuando termines un ejercicio, en lugar de simplemente pasar al siguiente, detente y piensa en estas preguntas:

1. ¿Por qué funcionó ese procedimiento?
2. ¿Qué podría cambiarse en la pregunta para que todavía funcione?

3. ¿Qué podría cambiarse en la pregunta para que *no* funcione?
4. ¿Podría modificar el procedimiento para que funcione en algunos de estos casos?

Todas estas preguntas deberían ayudarte a construir flexibilidad en la aplicación de lo que sabes.

Queremos volver a la idea de que saber cómo aplicar tales procedimientos es solo una parte de entender las matemáticas. Es una parte importante, pero la mayoría de los estudiantes pueden reconocer la diferencia entre aprender a aplicar un procedimiento mecánicamente y entender por qué funciona. Aprender mecánicamente tiene algunas ventajas: generalmente es rápido y relativamente sencillo. Pero también tiene desventajas: si aprendes procedimientos mecánicamente, es más fácil olvidarlos, aplicarlos incorrectamente y mezclarlos (y será más difícil para ti tener tanto éxito en este curso como te gustaría). Desarrollar una comprensión adecuada de por qué las cosas funcionan generalmente es más difícil y lleva más tiempo, pero el conocimiento resultante es más fácil de recordar y más favorable para un razonamiento flexible y preciso (y sin duda te ayudará a obtener una buena calificación en este curso).

Para resumir, antes de tomar cursos de nivel superior como cálculo y álgebra, debes repasar tu conocimiento de los procedimientos estándar porque tus profesores esperarán que seas capaz de usarlos con fluidez. A medida que avances, se esperará que asumas más responsabilidad para decidir qué procedimiento aplicar; podría ser una buena idea practicar esto trabajando en ejercicios de fuentes que no te digan exactamente qué hacer. También se espera que adaptes los procedimientos de manera sensata, y que determines cómo se pueden aplicar los teoremas o definiciones sin necesariamente haber visto muchos ejemplos resueltos. No tendrás éxito en matemáticas de nivel superior si siempre intentas resolver problemas buscando algo que se vea similar y copiándolo. Tienes que ser más reflexivo que eso. Las matemáticas no son solo procedimientos. La fluidez con los procedimientos es importante, pero en muchos casos deberías aspirar a una comprensión más profunda de por qué los procedimientos funcionan.

Definiciones

Nos encontraremos con muchas *definiciones*, *teoremas* y *demostraciones* a lo largo de cualquier asignatura de matemáticas en la universidad.

Una definición no tiene nada que ver con algo que sea verdadero. Solo nos dice lo que significa una palabra matemática. Una definición puede definir un tipo de objeto o puede definir una propiedad.

Ya sabes que una definición nos dice lo que significa una palabra porque has estado usando diccionarios durante mucho tiempo. Pero hay dos diferencias muy, muy importantes entre las definiciones de diccionario y las definiciones matemáticas. Si quieres entender el material en esta asignatura, es vital que entiendas estas diferencias.

La primera diferencia es que cuando los matemáticos (en particular yo, tu profesor de cálculo) enuncian una definición, **realmente lo dicen en serio**. No quieren decir que esta es una buena descripción de la mayoría de los casos, pero que podría haber excepciones en algún lugar. Esta **no** es la forma en que funcionan las definiciones de diccionario. Si tomaras dos diccionarios y buscaras un concepto cotidiano, ya sea uno concreto (como "mesa") o uno abstracto (como "justicia"), ¿esperarías que las definiciones fueran idénticas? Probablemente no. De hecho, probablemente esperarías poder encontrar cosas en el mundo que satisfagan una definición pero no la otra. O encontrar algo que quisieras llamar "mesa" o "justicia", pero

que realmente no satisfaga ninguna de las dos definiciones[1]. O encontrar algo para lo que, aunque haya una definición, la gente no esté de acuerdo.

Ahora, compara esto con lo que sucedería si tomaras un libro de matemáticas y buscaras una definición de "número par". ¿Esperarías poder encontrar un número par que no satisfaga la definición? ¿O un número no par que, sin embargo, satisfaga la definición? Absolutamente no. Las excepciones simplemente no existen. No es como si pudieras encontrar un número tan grande que pudiera ser par sin ser divisible por 2. Si tomaras dos libros de texto, quizás no esperarías que las dos definiciones se redactaran exactamente de la misma manera, pero sí esperarías que fueran lógicamente equivalentes. Es decir, esperarías que todas y solo las cosas que satisfacen la primera definición también satisfagan la segunda. Así que esa es una diferencia: las definiciones matemáticas significan exactamente, **exactamente** lo que dicen. No hay excepciones, y podrían existir diferentes redacciones, pero estas deben ser lógicamente equivalentes.

La otra diferencia, que es en parte una consecuencia de la primera, es que las definiciones matemáticas son precisas y operables de una manera que las definiciones de diccionario no lo son. Esto significa que contienen información que podemos manipular realmente en un argumento algebraico o lógico. Por ejemplo, considera la siguiente definición simple:

Definición: Un número es *par* si es divisible por 2.

"Bueno, sí", pensarás, "obviamente". Sin embargo, esta probablemente no es la forma en que tu profesor escribirá esta definición. Es más probable que veas algo como esto:

Definición: Un número n es *par* si y solo si existe un número entero k tal que $n = 2k$.

Podrías pensar que esto complica demasiado las cosas. Pero tiene algunas ventajas. La primera es la precisión. Podemos ver esto comparando con el tipo de cosas que los estudiantes a veces escriben cuando intentan definir *par*. Tienden a decir cosas como "Par es cuando es divisible por 2". Claramente, eso captura la idea clave, pero no es muy preciso. Para empezar, ¿qué es "es"? Claramente "es" se supone que es un número, pero este número no se presenta adecuadamente. En contraste, la mejor definición nos dice explícitamente que estamos tratando con un número y le da un nombre, n. Por otra parte, la definición del estudiante contiene la locución "es cuando". Esto tiende a sonar torpe, en matemáticas pero también en otros campos. En matemáticas, si te encuentras escribiendo "es" o "es cuando", probablemente deberías considerar reformular.

La segunda ventaja es la operatividad. Podemos operar con la definición matemática para demostrar cosas. La mejor definición nos da una forma de capturar y manipular números pares porque establece lo que significa la divisibilidad en términos algebraicos. Podríamos, por ejemplo, usarla para demostrar que cualquier múltiplo entero de un número par también debe ser par, tal vez escribiendo algo como esto:

Supongamos que n es un número par. Entonces (por definición) existe un número entero k tal que $n = 2k$. Ahora, sea z cualquier número entero y $y = zn$. Entonces $y = z(2k)$. Pero podemos reescribir esto como $y = 2(zk)$. Ahora zk es un número entero, por lo que y es par porque puede escribirse en la forma requerida.

[1] Por ejemplo, ¿es una "mesa" de 10 metros de altura en una exposición de arte una mesa? Más o menos, pero no cumpliría con criterios funcionales como ser una superficie plana sobre la que puedas descansar cosas: nadie podría alcanzar, y de todos modos probablemente no se permitiría tocar la exhibición.

Podríamos tener las mismas ideas si estuviéramos trabajando con la definición imprecisa del estudiante. Pero la mejor versión nos da una ventaja para escribir argumentos como este al proporcionar alguna notación. Además, esperamos que la definición precisa *excluya* todos los números que no son pares (para el número 3, no existe un número entero k apropiado). De esta manera, la definición de número par captura la noción de paridad de una manera razonable.

Un punto final aquí es que no "demostramos" definiciones. No podemos, porque no hay nada que demostrar: las definiciones simplemente capturan convenciones en las que todos acordamos usar una palabra para que signifique exactamente lo mismo. Un profesor podría, en algún momento, explicarte cómo una definición captura una idea intuitiva, pero esto no es lo mismo que demostrarla.

Teoremas

Mientras que una definición establece lo que queremos decir con una palabra que describe un objeto o propiedad matemática, un teorema nos habla de una relación entre dos o más tipos de objetos y propiedades; se asume que ya sabemos qué son esos objetos y propiedades. Por supuesto, si no conoces el significado de todas las palabras y símbolos en el enunciado de un teorema, probablemente sentirás que no entiendes el teorema. Sin embargo, ten en cuenta que, incluso si no entiendes un teorema, deberías ser capaz de reconocer que tiene una estructura típica de teorema de la forma:

Si esta cosa es cierta, *entonces* esta otra cosa también es cierta.

La parte que va con el *si* se llama la premisa, el supuesto o la hipótesis. La parte que va con el *entonces* se llama la conclusión. En la vida real, en realidad no es tan simple, porque hay algunas formas de formular teoremas que no hacen tan obvia la estructura "si... entonces...".

Los teoremas nos dicen cosas verdaderas sobre las relaciones entre conceptos. Sin embargo, necesitamos tomarnos la molestia de probarlos. A veces lo hacemos porque los teoremas están lejos de ser obvios. Otras veces, sin embargo, los teoremas son bastante obviamente verdaderos, y lo hacemos por una razón más sutil: demostrar teoremas nos permite ver cómo todos se interrelacionan para formar una teoría coherente.

Una vez que las definiciones están establecidas, los teoremas siguen por necesidad lógica. Por ejemplo, una vez que hemos decidido definir los números pares como números divisibles por 2, debemos concluir que cualquier múltiplo entero de un número par también es par. No tenemos opción respecto a ese tipo de consecuencia.

Debemos mencionar que, aunque solo hemos usado el término *teorema*, hay varias palabras que a veces se usan en su lugar. Algunas de estas son *proposición*, *lema*, *afirmación*, *corolario* y el aparentemente abarcador *resultado*. *Lema* generalmente se usa para un teorema pequeño que luego se usará para probar uno más grande e importante. *Corolario* se usa para un resultado que se sigue como una consecuencia bastante inmediata de un teorema importante. Los otros términos son más o menos intercambiables (en nuestra opinión).

Pensar en objetos puede permitirnos vincular nuevos enunciados con nuestro conocimiento existente. Sin embargo, también podemos trabajar hacia la comprensión de los enunciados pensando en sus estructuras

lógicas. Para hacerlo, necesitamos prestar atención a los usos matemáticos del lenguaje lógico. Comenzaremos examinando más cuidadosamente las formas en que se usa la palabra *si* en matemáticas.

Primero, considera un enunciado "Si A, entonces B". Esto a veces se escribe como "$A \implies B$", que se lee en voz alta como "A implica B". El uso de una flecha hace más claro que podemos pensar en este enunciado como si tuviera una dirección, lo cual es importante porque podríamos tener una situación en la que $A \implies B$ es un enunciado verdadero pero $B \implies A$ no lo es. A veces ambas implicaciones son válidas, como en el caso:

$$x \text{ es par} \implies x^2 \text{ es par (cierto)}$$
$$x^2 \text{ es par} \implies x \text{ es par (cierto)}$$

Sin embargo, a veces una es verdadera pero la otra no, como en el caso:

$$x < 2 \implies x < 5 \text{ (cierto)}$$
$$x < 5 \implies x < 2 \text{ (falso)}$$

Estos enunciados son en realidad algo imprecisos, porque no hemos especificado de qué tipo de objeto es x. Probablemente asumiste que debe ser un entero en el caso del cuadrado y un número real en el caso de las desigualdades, ya que eso tendría sentido. Pero, para ser más claros, podríamos escribir cosas como:

$$\text{Para cada } x \in \mathbb{R},\ x < 5 \implies x < 2.$$

Hacerlo facilita ver por qué este enunciado particular es falso: hay algunos números reales que son menores que 5 pero no menores que 2. Pero los matemáticos a veces omiten tales frases cuando escriben en forma de notas o cuando la interpretación pretendida es obvia.

Cuando quieren discutir ambas implicaciones a la vez, los matemáticos usan una flecha de doble punta que significa "es equivalente a", o escriben "si y solo si" o su abreviatura "sii". Así que estas son diferentes formas de escribir el mismo enunciado (verdadero) sobre números enteros:

$$x \text{ es par} \iff x^2 \text{ es par.}$$
$$x \text{ es par si y solo si } x^2 \text{ es par.}$$
$$x \text{ es par sii } x^2 \text{ es par.}$$

Hemos visto la frase "si y solo si" antes, en nuestras definiciones. Aquí está de nuevo:

Definición: Un número n es *par* si y solo si existe un entero k tal que $n = 2k$.

Para pensar en la frase, podría ser esclarecedor dividir esta definición y escribir cada implicación por separado:

Un número n es par *si* existe un entero k tal que $n = 2k$.

Un número n es par *solo si* existe un entero k tal que $n = 2k$.

¿Puedes ver cómo esta definición "atrapa" los números que son pares y excluye a los que no lo son?

Una última cosa a tener en cuenta sobre un enunciado de la forma "A si y solo si B" es que, si queremos demostrarlo, podemos tomar uno de dos enfoques. Podemos construir una demostración en la que todas las líneas sean equivalentes entre sí, o probar los dos enunciados $A \implies B$ y $B \implies A$ por separado.

Los usos de "si" y " $\implies$ " parecen sencillos cuando estamos considerando enunciados matemáticos simples. Pero nos gustaría llamar tu atención sobre dos posibles fuentes de confusión.

Primero, se necesita pensar un poco para ordenar cuál de los "si" y el "solo si" corresponde a cuál implicación. Deberías pensar en esto, tal vez pensando en cuál podría reemplazar la flecha de "implica" en estas dos versiones del mismo enunciado (verdadero):

$$x < 2 \implies x < 5 \quad x < 5 \impliedby x < 2.$$

En segundo lugar, resulta que los estudiantes *no* siempre interpretan "si" de manera matemática en la vida cotidiana. En la conversación cotidiana, tendemos a hablar de manera bastante imprecisa, confiando en el contexto para ayudar a nuestro oyente a hacer la interpretación que pretendemos. Por ejemplo, imagina que alguien te dice:

Si limpias el coche, entonces puedes salir el viernes por la noche.

Podrías inferir razonablemente de esto que si no limpias el coche, entonces no puedes salir el viernes por la noche. Claramente eso es lo que el hablante pretende. Y alguien más podría inferir que si se te permitió salir el viernes, entonces debes haber limpiado el coche. Pero, de hecho, *ninguno de estos es lógicamente equivalente al enunciado original.* Tal vez la forma más fácil de ver esto es mirar la lógica en paralelo con nuestros simples enunciados sobre desigualdades:

limpiar coche $\implies$ salir viernes	$x < 2 \implies x < 5$
no limpiar coche $\implies$ no salir viernes	$x \geq 2 \implies x \geq 5$
limpiar coche $\impliedby$ salir viernes	$x < 2 \impliedby x < 5.$

El segundo y tercer enunciado en cada caso no son lógicamente los mismos que el primero. Alternativamente, podrías pensar en la situación cotidiana y ver que el enunciado original no dice nada sobre lo que sucede si no limpias el coche, por lo que no habría contradicción si no lo limpias pero aún así se te permitiera salir. Técnicamente, la persona que negocia contigo debería decir:

Puedes salir el viernes por la noche *si y solo si* limpias el coche.

Por supuesto, nadie habla así. Lo cual significa que podrías tener menos práctica de lo que piensas interpretando correctamente las declaraciones lógicas. Usar el lenguaje lógico en un sentido matemáticamente correcto no es demasiado difícil, sin embargo, porque hay casos en los que la interpretación natural del lenguaje *sí* es la misma que la matemática. Considera la afirmación:

Si Juan es de Sevilla, entonces Juan es de Andalucía.

Nadie que escuche esto soñaría con inferir cualquiera de estas dos cosas:

Si Juan no es de Sevilla, entonces Juan no es de Andalucía.
Si Juan es de Andalucía, entonces Juan es de Sevilla.

Pero estas inferencias son análogas a las que vimos para la afirmación sobre limpiar el coche. Asegúrate de ver cómo.

En el caso de Sevilla, la interpretación natural de la afirmación es la misma que la interpretación matemática. También podemos usarlo para ilustrar un punto general sobre la equivalencia lógica de diferentes implicaciones. Para cualquier afirmación de la forma $A \implies B$, podemos considerar tres afirmaciones relacionadas llamadas su *conversa*, *inversa* y *contrarrecíproca*. Esto es lo que significa cada una, utilizando el ejemplo de Sevilla como ilustración.

Original	$A \implies B$	de Sevilla $\implies$ de Andalucía
Conversa	$B \implies A$	de Andalucía $\implies$ de Sevilla
Inversa	no $A \implies$ no B	no de Sevilla $\implies$ no de Andalucía
Contrarrecíproca	no $B \implies$ no A	no de Andalucía $\implies$ no de Sevilla.

Esto es lo que significa cada una usando uno de nuestros ejemplos matemáticos simples en su lugar:

Original	$A \implies B$	$x < 2 \implies x < 5$
Conversa	$B \implies A$	$x < 5 \implies x < 2$
Inversa	no $A \implies$ no B	$x \geq 2 \implies x \geq 5$
Contrarrecíproca	no $B \implies$ no A	$x \geq 5 \implies x \geq 2$.

Esto debería ayudarte a recordar que si $A \implies B$ es una afirmación verdadera, entonces su contrarrecíproca también será verdadera (de hecho, son lógicamente equivalentes), pero su inversa y su conversa podrían no serlo.

Finalmente, debo señalar que tu profesor será cuidadoso con el uso de "si" y "si y solo si" en los teoremas y demostraciones, pero quizás no sea tan cuidadoso en las definiciones. En una definición, podría simplemente escribir "si" en lugar de "si y solo si". Esto funciona por la misma razón por la que funciona la comunicación cotidiana: todos saben que, en una definición, esto es lo que se pretende.

A continuación, queremos hablar de las frases "para todo" y "existe". Estas se llaman *cuantificadores* porque nos dicen cuántos de algo estamos hablando. Son tan comunes en matemáticas que tenemos símbolos para ellos: usamos "$\forall$" (el cuantificador universal) para significar "para todo" y "$\exists$" (el cuantificador existencial) para significar "existe".

En declaraciones simples, los cuantificadores son fáciles de entender. Aquí hay una declaración simple con un cuantificador:

$$\forall x \in \mathbb{Z}, x^2 \geq 0.$$

En esta declaración, escribimos $\forall x \in \mathbb{Z}$ para especificar exactamente de qué objetos estamos hablando. Podríamos simplemente escribir $\forall x$, y algunas veces la gente lo hace cuando es obvio de qué tipo de números (u otros objetos) se trata una declaración. Pero hacerlo podría ser ambiguo. En este caso, la declaración podría igualmente referirse a números reales o complejos, lo que plantea problemas sobre la veracidad de la declaración: "$\forall x \in \mathbb{Z}, x^2 \geq 0$" es verdadera, "$\forall x \in \mathbb{C}, x^2 \geq 0$" no lo es. Por lo tanto, es una buena práctica ser específico.

Las declaraciones cuantificadas más complicadas pueden ser más difíciles de entender. A continuación, una definición escrita en palabras y en una forma abreviada usando el nuevo símbolo (que podría leerse de

manera más natural como "para cada" en este caso):

Definición: Una función $f : \mathbb{R} \to \mathbb{R}$ es *creciente* si y solo si, para cada $x_1, x_2 \in \mathbb{R}$ tales que $x_1 < x_2$, tenemos $f(x_1) \leq f(x_2)$.

Definición (abreviada): $f : \mathbb{R} \to \mathbb{R}$ es *creciente* si y solo si $\forall x_1, x_2 \in \mathbb{R}$ tales que $x_1 < x_2$, $f(x_1) \leq f(x_2)$.

Aquí hay una declaración simple cuantificada que involucra el cuantificador existencial:

$$\exists x \in \mathbb{Z} \text{ tal que } x^2 = 25.$$

Esta es una declaración verdadera, porque cuando los matemáticos dicen "existe", quieren decir "existe al menos uno". Aquí, hay dos enteros diferentes x que satisfacen la declaración. En otros casos, puede haber cientos. A los estudiantes a veces les resulta extraño que digamos "existe" sin especificar cuántos, porque, si sabemos exactamente cuántos hay, parece descortés no decirlo. Sin embargo, las matemáticas avanzadas tratan, al menos en parte, de las relaciones generales entre conceptos, en lugar de encontrar "respuestas" como tales. Podemos ver otras razones por las cuales tiene sentido usar "existe" sin especificación adicional si observamos otra definición (nuevamente mostrada tanto en palabras como en una forma abreviada):

Definición: Un número n es *par* si y solo si existe un entero k tal que $n = 2k$.

Definición (abreviada): $n \in \mathbb{Z}$ es *par* si y solo si $\exists k \in \mathbb{Z}$ tal que $n = 2k$.

Esta definición nos proporciona una forma acordada de decidir si un número es par o no. Para tomar esa decisión, no nos importa cuál sea el k particular, solo nos importa si existe o no uno. También nos permite hacer argumentos generales sobre todos los números pares. Podríamos comenzar diciendo "Supongamos que n es par, entonces $\exists k \in \mathbb{Z}$ tal que $n = 2k$". En este caso, no queremos especificar cuál es el k porque queremos que el argumento resultante sea general en el sentido de que se aplique a cualquier número que satisfaga la condición.

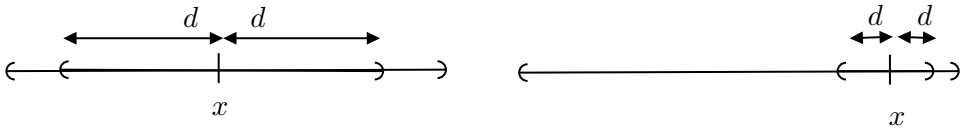
Algunas declaraciones matemáticas tienen más de un cuantificador. La siguiente definición (nuevamente con una versión abreviada) podría describirse como doblemente cuantificada o como con dos cuantificadores anidados:

Definición: Un conjunto $X \subseteq \mathbb{R}$ es *abierto* si y solo si para cada $x \in X$ existe $d > 0$ tal que $(x - d, x + d) \subseteq X$.

Definición (abreviada): $X \subseteq \mathbb{R}$ es *abierto* si y solo si $\forall x \in X \; \exists d > 0$ tal que $(x - d, x + d) \subseteq X$.

Cuando una declaración tiene más de un cuantificador, *el orden en el que aparecen es realmente importante.* En este caso, la definición dice: "para cada x, existe un d". Debemos imaginar tomar un valor particular de

x y encontrar un d adecuado (quizás dependiendo de x). Si tomamos un x diferente, podríamos necesitar un d diferente (tal vez uno más pequeño, como en el diagrama de la derecha a continuación).



Si en cambio la definición dijera "existe un d para cada x", los matemáticos interpretarían esto como que podemos seleccionar un solo d (independiente de x) que funcione para cada x. Eso no es lo mismo en absoluto.

Para comprobar tu comprensión de esto, considera las siguientes dos afirmaciones. Una es verdadera y la otra es falsa. ¿Cuál es cuál?

$$\exists y > 0 \text{ tal que } \forall x > 0,\ y < x.$$
$$\forall x > 0\ \exists y > 0 \text{ tal que } y < x.$$

Es normal encontrar esto difícil, porque en la vida cotidiana probablemente no distinguiríamos entre estas dos afirmaciones. Sin darnos cuenta, haríamos la interpretación que parece más realista, sin importar la corrección lógica. Por lo tanto, la mayoría de los estudiantes tienen que concentrarse un rato antes de acostumbrarse a leer lo que está literalmente allí y hacer la interpretación matemáticamente correcta.

Aunque hemos escrito los teoremas en la forma "Si... entonces...", también puedes ver otras formulaciones. Estas son algunas comunes:

Teorema: Si f es una función par, entonces $\dfrac{df}{dx}$ es una función impar.

Teorema: Supongamos que f es una función par. Entonces $\dfrac{df}{dx}$ es una función impar.

Teorema: Toda función par tiene una derivada impar.

Todas estas serían interpretadas como que significan lo mismo: la premisa en cada caso es que la función es par, y la conclusión es que su derivada es impar. Puede parecer extraño que no elijamos una forma y nos atengamos a ella, pero a veces una versión u otra suena más natural, así que a los matemáticos les gusta tener esta flexibilidad.

También verás teoremas de diferentes tipos. Hay, por ejemplo, *teoremas de existencia* como este:

Teorema: Existe un número x tal que $x^3 = x$.

Una forma de probar un teorema como este es simplemente producir un objeto que lo satisfaga: el número 1 serviría, en este caso. No siempre es tan fácil, pero es importante reconocer que podría serlo, porque a veces los estudiantes se complican haciendo cosas complejas cuando una respuesta simple sería suficiente.

También hay teoremas sobre la no existencia, como este:

Teorema: No existe un número primo máximo.

De hecho, con un poco de reflexión, los teoremas de no existencia pueden reformularse en nuestra forma estándar. Este, por ejemplo, podría escribirse con un cuantificador universal:

Teorema: Para todo número primo n existe un número primo p tal que $p > n$.

Luego podría reformularse en nuestra forma inicial:

Teorema: Si n es un número primo, entonces existe otro número primo p tal que $p > n$.

Estas posibilidades de reformulación pueden ser muy útiles cuando queremos empezar a probar algo: a veces reescribir de una manera diferente puede darnos diferentes ideas sobre cosas sensatas para intentar. Sin embargo, el hecho de que a menudo podamos reformular no significa que puedas ser descuidado con tu escritura matemática. El parafraseo descuidado puede cambiar fácilmente el significado lógico de una afirmación. Una vez que te vuelvas fluido en el uso del lenguaje lógico de manera matemática, encontrarás que puedes cambiar de formas sin alterar el significado. Hasta ese punto, deberías pensar cuidadosamente sobre la precisión lógica.

Una gran ventaja de tener significados precisos para los términos lógicos es que nos otorga mucho poder de razonamiento mecánico. Por ejemplo, si sabemos que $A \implies B$ y que $B \implies C$, entonces podemos deducir que $A \implies C$. Podemos hacer esto incluso si A, B y C son sobre objetos realmente complicados que nunca hemos conocido y que no entendemos. De manera similar, si quisiéramos probar una afirmación de la forma $A \implies B$, pero no estamos progresando mucho, podemos recordar que la contraposición (no $B \implies$ no A) es siempre equivalente a la original, y tratar de probar eso en su lugar.

Esto es lo que queremos decir cuando decimos que podemos desarrollar una comprensión valiosa observando la estructura lógica de una afirmación. Si prestamos atención a construcciones que involucran "si" o un cuantificador, podemos hacer uso de tales regularidades en nuestro razonamiento. Esto es (al menos en parte) lo que la gente quiere decir cuando habla de trabajo *formal*: podemos concentrarnos en la forma lógica de una oración e ignorar temporalmente su contenido significativo. No *tenemos que* ignorar el significado, por supuesto, y para la mayor parte de la discusión anterior probablemente estabas pensando también en los significados. Pero prestar atención a la forma lógica es vital para una comprensión adecuada. Algunos estudiantes son laxos en esto; cuando leen matemáticas, miran principalmente los símbolos, ignorando o pasando por alto las palabras. Esto puede hacer que su comprensión sea defectuosa y su escritura inexacta, porque confunden importantes cuantificadores o implicaciones. Por ejemplo, considera el siguiente teorema:

Teorema de Rolle: Supongamos que $f : [a, b] \to \mathbb{R}$ es continua en $[a, b]$ y diferenciable en (a, b) y que $f(a) = f(b)$. Entonces $\exists c \in (a, b)$ tal que $f'(c) = 0$.

Es bastante común ver a los estudiantes cometer errores, escribiendo cosas como esta:

Teorema de Rolle: Supongamos que $f : [a, b] \to \mathbb{R}$ es continua en $[a, b]$ y diferenciable en (a, b). Entonces $f(a) = f(b)$ y $\exists c \in (a, b)$ tal que $f'(c) = 0$.

Podemos ver que estos son diferentes al observar sus formas lógicas: una de las premisas en la versión correcta aparece en cambio como parte de la conclusión en la segunda (mira cuidadosamente para asegurarte de que puedes ver esto). Claramente, eso debe hacer una diferencia muy importante, por lo que la versión incorrecta no puede ser lógicamente equivalente a la versión correcta. Sin embargo, todavía podría ser un teorema válido. En este caso, sin embargo, no lo es, lo cual podemos ver pensando en términos de ejemplos. En la versión incorrecta, las premisas introducen una función f que está definida en un intervalo y es continua y diferenciable en este intervalo. La conclusión afirma que los valores de la función son iguales en los extremos del intervalo, pero esto no puede derivarse de manera válida de las premisas, porque hay muchas funciones e intervalos que satisfacen las premisas pero no tienen esta propiedad. Por ejemplo, $f(x) = x^2$ es continua en $[0, 2]$ y diferenciable en $(0, 2)$, pero ciertamente no es el caso que $f(0) = f(2)$. Podemos ver cómo un estudiante podría escribir la versión incorrecta en primer lugar, pero alguien que está pensando en el significado de su escritura debería reconocer tales errores al releer.

Esto nos lleva de nuevo a la idea de que tanto la forma lógica como los objetos de ejemplo pueden contribuir a la comprensión matemática, aunque centrarse en cada uno tiene diferentes ventajas y desventajas. Si te fijas principalmente en ejemplos, podrías sentir que entiendes, pero podrías no apreciar la plena generalidad de una afirmación o encontrar difícil ver la estructura lógica de todo un curso. Si te centras principalmente en argumentos formales, podrías ser capaz de ver cómo todo encaja lógicamente, pero podrías encontrarte quejándote de que es muy abstracto y que realmente no entiendes lo que está sucediendo. Tu experiencia con estos temas probablemente variará de un curso a otro, porque algunos profesores dan muchos ejemplos y dibujan muchos diagramas, mientras que otros dan una presentación mucho más formal. Si el enfoque de un profesor no coincide con tu forma preferida de desarrollar la comprensión, podrías encontrar útil fortalecer tu comprensión de los vínculos entre los objetos de ejemplo y el trabajo formal.

Demostraciones

Has estado construyendo demostraciones (o pruebas) matemáticas durante muchos años. Por ejemplo, para hacer los cálculos necesarios para demostrar que las soluciones de la ecuación $x^2 - 20x + 10 = 0$ son $10 + 3\sqrt{10}$ y $10 - 3\sqrt{10}$, escribirías algo como esto:

$$x = \frac{20 \pm \sqrt{400 - 40}}{2} = \frac{20 \pm \sqrt{360}}{2} = \frac{20 \pm 6\sqrt{10}}{2} = 10 \pm 3\sqrt{10}.$$

Este cálculo utiliza métodos que todos consideran válidos, por lo que captura todo lo que necesitamos para una demostración de que las soluciones son realmente las afirmadas. Para hacer que parezca más una demostración de nivel avanzado, podríamos reescribirlo así:

Afirmación: Si $x^2 - 20x + 10 = 0$ entonces $x = 10 + 3\sqrt{10}$ o $x = 10 - 3\sqrt{10}$.

Demostración: Supongamos que $x^2 - 20x + 10 = 0$. Entonces, usando la fórmula cuadrática,

$$x = \frac{20 \pm \sqrt{400 - 40}}{2} = 10 \pm 3\sqrt{10}.$$

Esta versión declara explícitamente la afirmación, comienza con la premisa ("Supongamos que $x^2 - 20x + 10 = 0$"), y contiene pocas palabras para justificar aquellos pasos que son más sofisticados o menos obvios.

Mi punto es que no hay nada inherentemente misterioso en las demostraciones. Es cierto que la mayoría de las matemáticas de la escuela secundaria no se presentan de esta manera, pero también es cierto que la mayoría de ellas podrían estarlo. Decimos esto porque, en los cursos de nivel superior, gran parte de las matemáticas que encuentres se presentarán de esta forma; las matemáticas de la universidad estarán llenas de teoremas y demostraciones. Esto puede parecer un cambio bastante abrupto, y algunos estudiantes tienen la idea de que la escritura de demostraciones es un arte oscuro y misterioso al que solo los muy privilegiados tienen acceso. No lo es. En casos como el anterior, todo lo que realmente implica es escribir de una manera más profesionalmente matemática, es decir, escribir menos como un estudiante y más como un libro de texto, por así decirlo.

Esto no es para menospreciar las dificultades genuinas que enfrentan los estudiantes universitarios al manejar demostraciones. Obviamente, el ejemplo anterior es simple, y las demostraciones que se te pedirán entender y construir en cursos de nivel avanzado a menudo (aunque no siempre) serán mucho más difíciles. Puede que te lleve un tiempo acostumbrarte a digerir matemáticas presentadas de esta manera, y a escribir tus propias matemáticas de manera más profesional. Pero no hay razón para pensar que no lo lograrás, y esta sección discute algunas cosas a las que podrías prestar atención para acostumbrarte rápidamente.

Una de las cosas que a menudo se te pedirá hacer es demostrar que un objeto matemático satisface una definición. Sin embargo, la pregunta no se formulará de esa manera. Simplemente dirá algo como: "Demuestra que el conjunto $(2, 5)$ es abierto". Tendrás que interpretar esto como "Demuestra que el conjunto $(2, 5)$ satisface la definición de conjunto abierto", y revisar la definición de conjunto abierto para asegurarte de que tienes claro lo que esto implica. Esto suena simple, pero a menudo hemos visto estudiantes que, enfrentados con una instrucción como "Demuestra que el conjunto $(2, 5)$ es abierto", no saben qué hacer. Si no sabes cómo empezar con un problema de demostración, tu primer pensamiento a menudo debería ser, ¿qué dice la definición?

Hemos visto la definición relevante, que dice:

Definición: Un conjunto $X \subseteq \mathbb{R}$ es *abierto* si y solo si $\forall x \in X$, $\exists d > 0$ tal que $(x - d, x + d) \subseteq X$.

¿Cómo deberíamos escribir una demostración de que $(2, 5)$ es un conjunto abierto? A menudo, el mejor consejo es seguir la estructura de la propia definición. Queremos demostrar que $(2, 5)$ es abierto, por lo que queremos mostrar que para cada $x \in (2, 5)$ existe un $d > 0$ tal que $(x - d, x + d) \subseteq (2, 5)$. Cuando queremos demostrar que algo es cierto para cada x en algún conjunto, usualmente comenzamos nuestra demostración introduciendo uno, así:

Afirmación: $(2, 5)$ es un conjunto abierto.

Demostración: Sea $x \in (2, 5)$ arbitrario.

Aquí, *arbitrario* significa que estamos tomando cualquier x sin propiedades específicas. No es necesario escribir esto — muchas personas simplemente escribirían "Sea $x \in (2,5)$" — pero enfatiza que el argumento que sigue funcionará para cualquier x en el conjunto.

Ahora necesitamos mostrar la existencia de un d apropiado. La forma más simple de mostrar la existencia de algo es producir uno. En este caso, necesitamos producir un d que funcione para nuestro x. Este d dependerá de x, y una forma de hacerlo es elegir d como el mínimo de las dos distancias $5-x$ y $x-2$ (piensa en por qué). Así que podríamos escribir el resto de nuestra demostración de esta manera:

Afirmación: $(2,5)$ es un conjunto abierto.

Demostración: Sea $x \in (2,5)$ arbitrario. Sea d el mínimo de $5-x$ y $x-2$. Entonces $(x-d, x+d) \subseteq (2,5)$. Así que $\forall x \in (2,5)$, $\exists d > 0$ tal que $(x-d, x+d) \subseteq (2,5)$. Por lo tanto, $(2,5)$ es un conjunto abierto.

Observa que esta demostración refleja el orden y la estructura de la definición. Estamos mostrando algo para todo x, así que comenzamos con uno arbitrario. Estamos mostrando que para este x, existe un d apropiado, así que producimos uno. Debido a esto, la estructura de la demostración será obvia para un matemático, por lo que realmente no necesitas escribir la línea "Entonces $\forall x \in (2,5)\ldots$", pero podrías encontrarla útil para tu propio pensamiento.

Algunas personas tienden a incluir diagramas junto con demostraciones como esta, y otras no. Eso es porque los diagramas pueden ser esclarecedores, pero no son necesarios, y no son un sustituto para escribir una demostración correctamente (los diagramas no son demostraciones); los matemáticos (en particular tus profesores de álgebra y cálculo) quieren ver un argumento escrito que esté claramente vinculado a la definición apropiada.

Otro tipo estándar de demostración se conoce como *demostración directa*. En una demostración directa comenzamos suponiendo que las premisas son verdaderas y avanzamos, mediante una secuencia de manipulaciones válidas o deducciones lógicas, hasta la conclusión deseada. Aquí hay algunos teoremas para los que ya hemos estudiado una demostración directa:

Teorema: Si n es un número par, entonces cualquier múltiplo entero de n es par.

Teorema: Si $x^2 - 20x + 10 = 0$ entonces $x = 10 + 3\sqrt{10}$ o $x = 10 - 3\sqrt{10}$.

Teorema: $(2,5)$ es un conjunto abierto.

Detente y piensa por un momento aquí. ¿Puedes escribir las demostraciones de estas afirmaciones sin mirar? Si no puedes hacerlo de inmediato, ¿puedes recordar la esencia de cómo procedemos en cada caso y reconstruir el resto? Si te das un minuto para cada una, apostamos a que puedes recordar más de lo que inicialmente hubieras pensado. Los estudiantes a menudo tienen muy poca fe en su propia capacidad para recordar ideas matemáticas y reconstruir argumentos alrededor de ellas.

Una nota importante es que cuando hablamos de una demostración *directa*, nos referimos a la demostración final que escribimos, no necesariamente al proceso de construcción de esa demostración. Es posible

que puedas escribir las premisas y simplemente seguir tu intuición hasta llegar a una prueba, pero lo más probable es que tengas que probar algunas cosas como: escribir todo en términos de definiciones, pensar en algunos ejemplos, quizás dibujar un diagrama, etc. Luego, sin embargo, debes trabajar en cómo escribir tu prueba final de manera que quede clara su estructura lógica para un lector. Es probable que sea una buena idea tratar esta escritura como una tarea separada, digna de tu atención más allá de simplemente llegar a una respuesta o una demostración.

Una segunda cosa a tener en cuenta es que las demostraciones directas pueden tener estructuras internas algo más complicadas. La estructura más obvia de este tipo se da en una *demostración por casos*, que significa lo que parece: dividimos los casos que estamos tratando en grupos razonables y trabajamos con cada uno por separado dentro de la demostración principal. Considera el siguiente teorema:

Teorema: Para todo $x, y \in \mathbb{R}$, $\text{máx}\{x, y\} = \dfrac{x + y + |x - y|}{2}$.

Por supuesto, necesitamos definir los símbolos que aparecen en el enunciado.

Definición: Para todo $x, y \in \mathbb{R}$,

$$\text{máx}\{x, y\} = \begin{cases} x, & \text{si } x \geq y, \\ y, & \text{si } y < x. \end{cases}$$

Definición: Para todo $x \in \mathbb{R}$,

$$|x| = \begin{cases} x, & \text{si } x \geq 0, \\ -x, & \text{si } x < 0. \end{cases}$$

La primera definición le da significado a $\text{máx}\{x, y\}$ como "escoge el número más grande entre x e y", y la segunda definición es simplemente la función de valor absoluto usual. El teorema afirma que máx puede expresarse en términos de la función de valor absoluto. Para demostrar el teorema, debes reinterpretarlo de la siguiente manera:

Teorema: Para todo $x, y \in \mathbb{R}$, la expresión $\dfrac{x + y + |x - y|}{2}$ satisface la definición de $\text{máx}\{x, y\}$.

Observa que la definición de $\text{máx}\{x, y\}$ se da en un formato por partes, donde dos casos son obvios: $x \geq y$ e $y > x$. La función de valor absoluto también está definida en un formato por partes (y probablemente es la primera vez que la ves escrita formalmente de esta manera). Esto debería sugerirte considerar una prueba por casos. Antes de leer la prueba a continuación, deberías intentar llegar a una por ti mismo.

Demostración: *Caso 1:* Supongamos que $x \geq y$. Entonces $x - y \geq 0$ y, por lo tanto, por la definición de la función de valor absoluto, $|x - y| = x - y$. Por lo tanto,

$$\frac{x + y + |x - y|}{2} = \frac{x + y + x - y}{2} = \frac{2x}{2} = x.$$

Caso 2: Supongamos que $x < y$. Entonces $x - y < 0$ y, por lo tanto, por la definición de la función de valor absoluto, $|x - y| = -(x - y) = y - x$. Por lo tanto,

$$\frac{x + y + |x - y|}{2} = \frac{x + y + y - x}{2} = \frac{2y}{2} = y.$$

Juntando ambos casos, tenemos que para todo $x, y \in \mathbb{R}$,

$$\frac{x+y+|x-y|}{2} = \begin{cases} x, & \text{si } x \geq y, \\ y, & \text{si } y < x. \end{cases}$$

Por lo tanto, $\dfrac{x+y+|x-y|}{2}$ satisface la definición de máx$\{x, y\}$.

Entonces, ¿cuándo debes considerar una prueba por casos? A veces verás que no tienes más opción. En esta ilustración, por ejemplo, no tenemos muchas opciones; los valores de máx$\{x, y\}$ son diferentes dependiendo de si $x \geq y$ o $x < y$, así que tenemos que manejar estos casos por separado. En otras situaciones, puede que no sea necesario usar una prueba por casos, pero podría ser conveniente de todos modos porque hay algún tipo de división natural, quizás entre números positivos y negativos (como en la definición de la función de valor absoluto), o entre pares e impares. Finalmente, podría valer la pena comenzar una prueba por casos si crees que puedes construir un argumento para algunos de los objetos a los que se aplica el teorema pero no para otros. Un comienzo es mejor que nada y, una vez que tengas un argumento escrito para un caso, es posible que al reflexionar sobre él se te ocurran ideas sobre cómo continuar.

Un último consejo es que, si has producido una prueba por casos, o si estás mirando una producida por alguien más, podría ser una buena idea preguntarse si se podría reducir el número de casos. No tienes que hacer esto, por supuesto - si tu prueba es válida como está, está bien. Pero recuerda que los matemáticos también valoran la elegancia, y la brevedad contribuye a la elegancia, por lo que es un objetivo valioso.

El siguiente tipo de prueba del que queremos hablar es la *demostración por contradicción*. Este es un tipo de prueba indirecta, llamada así porque no procedemos directamente de las premisas a la conclusión. En cambio, hacemos una suposición temporal de que nuestra conclusión deseada (o alguna parte de ella) es falsa, y mostramos que esto nos lleva a una contradicción. De esto podemos deducir que la suposición temporal debe haber sido errónea, y por lo tanto que la conclusión deseada es verdadera. Esto suena bastante complicado, pero estás acostumbrado a hacer este tipo de argumento informalmente en la vida cotidiana. Aquí tienes un caso sencillo:

> Tu amigo: Daniel estuvo en casa en Vicálvaro todo el fin de semana.
>
> Tú: No, lo vi en Alcorcón el sábado por la tarde.

Aquí estás usando implícitamente una prueba por contradicción. La suposición temporal es que Daniel estaba en Vicálvaro. Tu argumento dice que si hacemos esa suposición, entonces podemos deducir que no estaba en Alcorcón (si quieres, esto usa el "teorema" de que la gente no puede estar en dos lugares a la vez). Pero esto contradice el hecho de que lo viste en Alcorcón. Así que la suposición de que estaba en Vicálvaro debe haber sido errónea.

A continuación, veremos un ejemplo matemático, que involucra la definición de *número racional*. La notación $\mathbb{Q}$ denota el conjunto de todos los números racionales y $\mathbb{Z}$ denota el conjunto de números enteros. Introducimos esta definición aquí:

Definición: $x \in \mathbb{Q}$ si y solo si $\exists p, q \in \mathbb{Z}$ (con $q \neq 0$) tal que $x = p/q$.

El teorema y la demostración a continuación usan el símbolo $\notin$ para significar "no es un elemento de", y en este caso $y \notin \mathbb{Q}$ significa que y es un *número irracional*. El teorema y la demostración asumen implícitamente que todos los números con los que estamos trabajando son reales (esto es común en el trabajo temprano con números racionales e irracionales). Como con cualquier teorema y demostración, debes leer todo cuidadosamente, asegurándote de entender lo que está pasando en cada paso.

Teorema: Si $x \in \mathbb{Q}$ e $y \notin \mathbb{Q}$ entonces $x + y \notin \mathbb{Q}$.

Demostración: Sea $x \in \mathbb{Q}$, entonces $\exists p, q \in \mathbb{Z}$ (con $q \neq 0$) tal que $x = p/q$. Sea $y \notin \mathbb{Q}$. Supongamos por contradicción que $x + y \in \mathbb{Q}$. Esto significa que $\exists r, s \in \mathbb{Z}$ (con $s \neq 0$) tal que $x + y = \frac{r}{s}$. Pero entonces

$$y = \frac{r}{s} - x = \frac{r}{s} - \frac{p}{q} = \frac{rq - ps}{sq}.$$

Ahora $rq - ps \in \mathbb{Z}$ y $sq \in \mathbb{Z}$ porque $p, q, r, s \in \mathbb{Z}$. También $sq \neq 0$ porque $q \neq 0$ y $s \neq 0$. Por lo tanto $y \in \mathbb{Q}$. Pero esto contradice la premisa del teorema. Por lo tanto, debe ser el caso que $x + y \notin \mathbb{Q}$.

En esta demostración, la suposición temporal es esta:

Supongamos por contradicción que $x + y \in \mathbb{Q}$.

Hacer esa suposición temporal nos lleva, mediante un uso razonable de definiciones y álgebra, a esta línea:

Por lo tanto, $y \in \mathbb{Q}$.

Esto (como se afirma) contradice la premisa del teorema, por lo que nos permite concluir que nuestra suposición temporal debe haber sido errónea, como esto:

Por lo tanto, debe ser el caso que $x + y \notin \mathbb{Q}$.

Debe quedar claro que para comprender adecuadamente una demostración como esta, debes hacer más de lo que habrías tenido que hacer en matemáticas anteriores. En la escuela secundaria, la mayor parte de tu lectura matemática habrá consistido en verificar algo de álgebra. Aquí, tienes que ser más sofisticado. Ciertamente debes comprobar para asegurarte de que puedes ver cómo funciona el álgebra y que no hay errores. Pero eso no es realmente donde está la acción en una demostración como esta. Para entenderla completamente, necesitas entender su estructura global. Debes ser capaz de identificar qué suposiciones se hacen dónde, identificar dónde surge la contradicción y qué es exactamente lo que se contradice, y comprender cómo todo encaja para demostrar que el teorema es verdadero.

El último tipo estándar de prueba que queremos discutir es la *prueba por inducción*. Dependiendo de tu experiencia previa, es posible que ya te hayas encontrado con esto. Si es así, lo habrás usado para demostrar cosas como esta:

$$P(n) \qquad \sum_{i=1}^{n} i^2 = \frac{n\,(n+1)\,(2n+1)}{6}.$$

Si no, es posible que no hayas visto esta notación, así que aquí hay una explicación rápida. El símbolo $\sum$ se llama "sigma" y es una letra griega mayúscula S, utilizada aquí para denotar una suma de $i = 1$ a $i = n$. Escrito de forma extendida, el lado izquierdo de lo anterior significa

$$\sum_{i=1}^{n} i^2 = 1^2 + 2^2 + 3^2 + \cdots + n^2.$$

Lo que hemos hecho aquí es sustituir $i = 1$, luego $i = 2$, luego $i = 3$, y así sucesivamente, hasta $i = n$, donde nos detenemos.

Por lo tanto, nuestra expresión original en realidad captura infinitas proposiciones:

$$P(1) \quad 1^2 = \frac{1\,(1+1)\,(2+1)}{6}$$

$$P(2) \quad 1^2 + 2^2 = \frac{2\,(2+1)\,(4+1)}{6}$$

$$P(3) \quad 1^2 + 2^2 + 3^2 = \frac{3\,(3+1)\,(6+1)}{6}$$

$$P(4) \quad 1^2 + 2^2 + 3^2 + 4^2 = \frac{4\,(4+1)\,(8+1)}{6}, \quad \text{y así sucesivamente.}$$

Tener un teorema que captura infinitos casos no es inusual; muchos de los otros teoremas que hemos visto hacen lo mismo. La diferencia aquí es que la forma del enunciado nos permite poner las proposiciones en una lista ordenada $P(1), P(2), P(3), P(4), \ldots$

La prueba por inducción funciona de la siguiente manera. Primero demostramos $P(1)$. Esto suele ser fácil. Luego hacemos algo inteligente. No intentamos demostrar ninguna de las otras proposiciones directamente. En cambio, tomamos un número general k y demostramos que si $P(k)$ es verdadero, entonces $P(k+1)$ también debe ser verdadero. Esto nos da $P(1) \implies P(2)$ y, como ya hemos demostrado $P(1)$, podemos concluir que $P(2)$ también es verdadero. También nos da $P(2) \implies P(3)$, por lo que podemos concluir que $P(3)$ también es verdadero. Ya entiendes la idea. Hemos hecho una cadena infinita de proposiciones, que son todas verdaderas porque la primera es verdadera y las implicaciones son todas verdaderas:

$$P(1) \implies P(2) \implies P(3) \implies P(4) \implies \cdots.$$

La prueba por inducción es una de esas ideas que los estudiantes suelen encontrar intuitivamente sencillas cuando se explica en abstracto. Sin embargo, a menudo les resulta difícil de usar en cualquier caso particular, por lo que analizaremos detenidamente el ejemplo con el que comenzamos. En ese ejemplo, es fácil demostrar que $P(1)$ es verdadero:

$$1^2 = 1 \quad \text{y} \quad 1 = \frac{1\,(1+1)\,(2+1)}{6}.$$

Demostrar que la implicación $P(k) \implies P(k+1)$ es un poco más difícil. Nos gustaría asumir que $P(k)$ es verdadero y usar esto para demostrar que $P(k+1)$ también es verdadero. Comenzaríamos haciendo algún trabajo preliminar en este punto, escribiendo algo como esto:

Asumiremos $P(k)$, lo que significa $\sum_{i=1}^{k} i^2 = \frac{k\,(k+1)\,(2k+1)}{6}$.

Queremos demostrar $P(k+1)$, lo que significa

$$\sum_{i=1}^{k+1} i^2 = \frac{(k+1)\,([k+1]+1)\,(2[k+1]+1)}{6},$$

lo que, al reescribir el lado izquierdo, significa que queremos

$$\left(\sum_{i=1}^{k} i^2\right) + (k+1)^2 = \frac{(k+1)\,([k+1]+1)\,(2[k+1]+1)}{6},$$

lo que, por la suposición sobre $P(k)$, significa que queremos

$$\frac{k\,(k+1)\,(2k+1)}{6} + (k+1)^2 = \frac{(k+1)\,([k+1]+1)\,(2[k+1]+1)}{6}.$$

Entonces solo tengo que hacer un poco de álgebra para mostrar que las últimas dos cosas son, de hecho, iguales (puedes intentar hacerlo).

Sin embargo, esta es definitivamente una situación en la que la forma en que piensas construir una prueba no es necesariamente la misma que la forma en que debes escribirla. El pensamiento anterior es completamente lógico, pero presentarlo de esa manera no funcionaría muy bien porque no coincide con la estructura de lo que estamos tratando de demostrar. Cuando escribimos una prueba de que $P(k) \implies P(k+1)$, realmente queremos que nuestra prueba comience con una clara suposición de $P(k)$ y proceda a través de una deducción clara y ordenada hasta $P(k+1)$.

Para las pruebas por inducción, favorecemos un diseño que haga que esa estructura sea muy clara. Escribiríamos algo como esto:

Teorema: $\forall n \in \mathbb{N}$, $\sum_{i=1}^{n} i^2 = \frac{n\,(n+1)\,(2n+1)}{6}$.

Demostración (por inducción): Sea $P(n)$ el enunciado $\sum_{i=1}^{n} i^2 = \frac{n\,(n+1)\,(2n+1)}{6}$.

Obsérvese que $1^2 = 1 = \frac{1\,(1+1)\,(2+1)}{6}$ por lo que $P(1)$ es verdadero.

Ahora sea $k \in \mathbb{N}$ arbitrario y supongamos que $P(k)$ es verdadero, es decir, que

$$\sum_{i=1}^{k} i^2 = \frac{k\,(k+1)\,(2k+1)}{6}.$$

Entonces

$$\begin{aligned}
\sum_{i=1}^{k+1} i^2 &= \left(\sum_{i=1}^{k} i^2\right) + (k+1)^2 \\
&= \frac{k\,(k+1)\,(2k+1)}{6} + (k+1)^2 \quad \text{(por la suposición)} \\
&= \frac{k\,(k+1)\,(2k+1) + 6(k+1)^2}{6} \\
&= \frac{(k+1)(2k^2+7k+6)}{6} \\
&= \frac{(k+1)\,(k+2)\,(2k+3)}{6} \\
&= \frac{(k+1)\,([k+1]+1)\,(2[k+1]+1)}{6}.
\end{aligned}$$

Por lo tanto $\forall k \in \mathbb{N}$, $P(k) \implies P(k+1)$. En consecuencia, por inducción matemática, $P(n)$ es verdadero $\forall n \in \mathbb{N}$.

Hay un par de cosas a notar sobre esto. Primero, la demostración contiene solo unas pocas palabras, pero estas ayudan a que la estructura sea clara. Segundo, todo el álgebra está en una sola cadena de igualdades que comienza con el lado izquierdo del enunciado de $P(k+1)$ y termina con el lado derecho. Puedes pensar por qué tiene sentido hacer las manipulaciones en este orden, dado que sabemos lo que buscamos. También puedes pensar por qué la última expresión de la cadena no es necesaria, pero podría ser útil para un lector que quiera vincular la prueba al teorema.

La confusión tiende a surgir con las demostraciones por inducción porque hay muchas cosas en las que pensar. Descubrimos que los estudiantes a menudo se confunden en el punto en el que escribimos: "Supongamos que $P(k)$ es verdadero". Los estudiantes suelen leer esto y pensar: "Pero eso es lo que queremos demostrar, ¿cómo es que se nos permite suponerlo?" De hecho, en esa etapa de la demostración, no estamos demostrando que $P(k)$ es verdadero, estamos demostrando que $P(k) \implies P(k+1)$. Asegúrate de ver la diferencia. Además, los estudiantes a veces se confunden, generalmente porque han permitido que ambigüedades se filtren en su redacción al usar la palabra "eso" en frases como "así que eso verdadero para n". Hay muchos posibles candidatos para el significado de "eso" en una demostración típica por inducción, por lo que deberías ser más específico. Escribir "Así que $P(n)$ es verdadero" es una forma de hacerlo. (Observa que la demostración anterior no incluye la palabra "eso"; somos muy específicos sobre lo que hemos deducido en cada etapa).

Entonces, ¿cuándo deberías usar una demostración por inducción? En algunos casos esto será obvio, porque es probable que tengas una sección sobre esto en al menos un curso. También te encontrarás con casos en los que deseas demostrar que algo es verdadero para todo $n \in \mathbb{N}$, lo cual es una pista de que vale la pena intentar la inducción. Sin embargo, ten en cuenta que los problemas para los cuales la inducción es útil pueden variar bastante. Primero, no hay ninguna razón particular para que una demostración por inducción comience en $n = 1$. Podrías ser solicitado a demostrar que algo es verdadero para cada $n \in \mathbb{N}$ tal que $n > 4$, por ejemplo. En ese caso, simplemente puedes hacer que $P(5)$ sea tu caso base y proceder como antes, excepto que en algún punto, tal vez en el *paso de inducción* (el punto en la demostración donde aplicas la suposición de que $P(k)$ es verdadero, conocido como la *hipótesis de inducción*), encontrarás que necesitas $n > 4$ para

justificar alguna manipulación que deseas hacer. Segundo, aunque algunos de los primeros problemas que encuentres involucren trabajar con una suma, la demostración por inducción es útil para muchos otros tipos de problemas. Todo lo que realmente necesitamos es una situación en la que tengamos infinitas declaraciones que puedan ser enumeradas en el orden de los números naturales, lo cual puede suceder de muchas maneras. Por ejemplo, considera estas tareas:

Demuestra que para cada número natural $n > 10$, $2^n > n^3$.

Demuestra que para todo $n \in \mathbb{N}$, $5^{3n} + 2^{n+1}$ es divisible por 3.

Para la primera tarea, escribiríamos

Sea $P(n)$ la afirmación de que $2^n > n^3$.

Luego probaríamos directamente que $P(11)$ es verdadero, es decir, que $2^{11} > 11^3$. Después, trabajaríamos para demostrar que si $k > 10$ y $2^k > k^3$, entonces $2^{k+1} > (k+1)^3$.

Para la segunda tarea, escribiríamos

Sea $P(n)$ la afirmación de que $5^{3n} + 2^{n+1}$ es divisible por 3.

Luego probaríamos directamente que $P(1)$ es verdadero, es decir, que $5^3 + 2^2$ es divisible por 3. Después, trabajaríamos para demostrar que si $5^{3k} + 2^{k+1}$ es divisible por 3, entonces $5^{3(k+1)} + 2^{(k+1)+1}$ es divisible por 3. Observa, en este caso, que la afirmación $P(k)$ **no** es simplemente "$5^{3k} + 2^{k+1}$". De hecho, $5^{3k} + 2^{k+1}$ no es una afirmación en absoluto; no podríamos demostrarlo porque es solo una expresión (para cualquier k en particular, es un número; no puedes "demostrar" un número, y no tiene sentido decir que un número implica otro). La afirmación es "$5^{3k} + 2^{k+1}$ es divisible por 3".

En nuestra experiencia, comenzar con una declaración clara de $P(n)$ a menudo marca la diferencia entre el éxito y el fracaso al construir una demostración por inducción, especialmente cuando se trata de un tipo de problema nuevo. Esto no es tan sorprendente, por supuesto: antes de comenzar cualquier problema, siempre debes asegurarte de tener claro lo que estás tratando de hacer. En cualquier caso, no siempre te dirán qué método usar, por lo que debes estar atento a casos menos familiares como estos, y debes entrenarte para notar cuándo la demostración por inducción podría ser útil.

Algunos consejos

Anteriormente dijimos que un estudiante nunca debe sentarse frente a un problema y pensar: "No sé qué hacer". Siempre hay cosas que intentar y, para ser un buen estudiante, debes estar dispuesto a intentarlas. De hecho, debes estar dispuesto a intentar cosas que resulten no funcionar. En nuestra experiencia, a veces los estudiantes no están dispuestos a hacer esto, por tres razones principales.

Primero, a algunos estudiantes no les gusta la inseguridad de no saber exactamente qué hacer. Eso los pone nerviosos. Quieren saber de antemano qué va a funcionar, y a veces piden la garantía del profesor sobre esto ("¿Es esta la forma correcta de hacerlo?"). El problema de buscar siempre esta garantía es que nunca descubres qué podrías haber hecho si lo hubieras intentado, lo que significa que nunca obtienes más confianza, lo que te lleva a un círculo vicioso, teniendo que pedir apoyo todo el tiempo.

Segundo, algunos estudiantes no quieren perder tiempo. Entendemos esto: obviamente nadie quiere pasar mucho tiempo en una sola cosa, especialmente cuando hay tantas cosas interesantes que hacer en la universidad. Pero es un gran error pensar que intentar algo que no funciona es una pérdida de tiempo. El tiempo dedicado a aprender nunca es tiempo perdido. Si intentas un método que no funciona, siempre que reflexiones sobre ello, aprendes *por qué* no funciona, lo que significa que sabes algo nuevo sobre la aplicabilidad del método. Y podrías ganar alguna idea sobre el problema para tener una mejor idea de qué intentar a continuación. Por supuesto, es un error seguir insistiendo en un método que claramente no está funcionando; la investigación muestra que los buenos solucionadores de problemas se detienen con frecuencia para reevaluar si su enfoque actual parece estar llevándolos a algún lado. Pero es un error aún mayor no empezar.

Tercero, algunos estudiantes no quieren desordenar su papel. Quieren saber que una vez que comiencen a escribir, podrán seguir escribiendo y llegar a una solución correcta, ordenada y prolija. Si esto te aplica, entonces nos tememos que tendrás que superarlo. El verdadero pensamiento matemático no es ordenado. Está lleno de comienzos en falso e intentos parciales y de realizaciones de que lo que parece no estar funcionando aquí sería, de hecho, una parte útil de una solución si se juntara con algo que falló hace diez minutos, o ayer, o la semana pasada. Es muy importante aceptar esto si deseas seguir mejorando como solucionador de problemas matemáticos. Necesitas tener intentos de soluciones parciales en papel por la simple razón práctica de que tu cerebro no puede manejar muchas cosas a la vez. Tienes una enorme cantidad de conocimiento almacenado en lo que se conoce como tu memoria a largo plazo, pero tu memoria de trabajo, donde realmente haces el nuevo pensamiento, tiene una capacidad seriamente limitada. No será lo suficientemente grande como para contener toda la información sobre un problema matemático complicado mientras simultáneamente trabajas en cómo resolverlo. Cuando escribes definiciones, teoremas o cálculos que podrían ayudarte a resolver un problema o construir una demostración, estás usando el papel para complementar tus capacidades cognitivas, extendiendo efectivamente la capacidad de tu memoria de trabajo. Así que no te preocupes por escribir cosas que están mal o que resultan no ser útiles. Siempre puedes redactar una versión ordenada de tu solución o demostración más tarde.

Tus profesores explicarán demostraciones que son largas y lógicamente complicadas. También explicarán demostraciones que dependen de alguna idea realmente ingeniosa. Esto tiende a preocupar a los estudiantes. Piensan: "Bueno, de acuerdo, puedo ver cómo funciona eso, pero *yo nunca habría pensado en eso*". Esto los hace preguntarse si son lo suficientemente buenos en matemáticas. Pero no deberías preocuparte, porque no se supone que puedas reinventar toda la matemática moderna teniendo todas las ideas originales tú mismo. Incluso a un estudiante de doctorado en matemáticas no se le esperaría tener muchas ideas totalmente originales. Como estudiante de grado, cuando te enfrentas a una demostración como esta, tu tarea es apreciar la idea ingeniosa, entender por qué funciona, pensar en cómo las modificaciones de la misma podrían funcionar en circunstancias ligeramente diferentes, y relacionarla con ideas utilizadas en otras partes del curso o en tu especialidad. Para tranquilizarte más, aquí hay una lista de cosas que *se espera* que hagas.

Primero, se espera que realices cálculos matemáticos rutinarios similares a los que has visto en matemáticas de nivel inferior. Como dijimos al principio de esta introducción, debes estar preparado para que estos cálculos sean más largos y más complicados que los que has experimentado antes, y debes estar preparado para tener que adaptar el procedimiento de cálculo si un paso en él no es válido para un nuevo caso.

Segundo, se espera que adaptes demostraciones que has visto a casos estrechamente relacionados. Por ejemplo:

- Habiendo visto la demostración de que $(2, 5)$ es un conjunto abierto, se podría esperar que demuestres

que cualquier intervalo de la forma (a, b) es un conjunto abierto.

- Habiendo visto la demostración de que $\max\{x, y\} = \dfrac{x + y + |x - y|}{2}$, se podría esperar que encuentres una fórmula, usando el valor absoluto, para expresar la función

$$\phi(x) = \begin{cases} x, & \text{si } x \geq 0, \\ 0, & \text{si } x < 0. \end{cases}$$

En tales casos, a menudo podrás tratar la demostración que has visto como un modelo, y cambiar algunos números de manera adecuada. Sin embargo, para reiterar uno de los puntos principales de esta introducción, no deberías hacer esto sin pensar; debes asegurarte de que cada paso de la demostración realmente funcione para los nuevos casos, y estar listo para hacer ajustes menores si no es así. Es importante tener cuidado en casos donde algún número podría ser cero, por ejemplo, o cuando divides ambos lados de una desigualdad por un número que podría ser negativo.

Tercero, se espera que adaptes demostraciones que has visto a casos que están relacionados, pero no tan estrechamente. Por ejemplo:

- Habiendo visto la demostración de que para todo $n \in \mathbb{N}$, $\displaystyle\sum_{i=1}^{n} i^2 = \frac{n\,(n+1)\,(2n+1)}{6}$, se podría esperar que demuestres que $a^n - b^n = (a - b)\displaystyle\sum_{i=1}^{n} a^{n-i} b^{i-1}$.
- Habiendo visto la demostración de que $(2, 5)$ es un conjunto abierto, se podría esperar que demuestres que $A = \{x \in \mathbb{R} : \ x^2 - 1 < 0\}$ es abierto.

En casos como estos, la demostración que has visto será ciertamente útil, pero no podrás tratarla como un modelo. Podrías ser capaz de construir una demostración que sea muy similar en su estructura básica, pero tendrás que pensar más para averiguar exactamente qué necesita cambiarse.

Cuarto, se espera que demuestres que las definiciones se cumplen. A veces se espera que hagas esto con definiciones que no has visto antes, si tu profesor cree que son lo suficientemente sencillas.

Quinto, se espera que construyas demostraciones de teoremas para los cuales no has visto un modelo estrechamente relacionado, o que resuelvas problemas que no has visto a tu profesor resolver en clase. Como hemos dicho, el mayor error que podrías cometer aquí sería sentarte pensando: "No nos han mostrado cómo hacer esto". Nadie te pedirá que hagas cosas que estén completamente fuera de tu alcance.

Sexto y finalmente, en un examen podrías ser solicitado a resolver algunos problemas desafiantes. A veces, una pregunta podría guiarte a través de una solución en pasos, o podría ofrecer una pista bastante grande para ayudarte con una idea clave o truco útil. A veces, podría simplemente preguntar de forma directa, lo que significa que tendrás que ser capaz de recordar las ideas clave o los trucos útiles tú mismo y hacer el resto. Para hacerlo, necesitarás haber leído y comprendido efectivamente el material en tus apuntes de clase.

Capítulo 4

El One-time pad y el Principio de Kerckhoffs

No puedes aprender sobre criptografía sin conocer a Alice, Bob y Eva. Este capítulo trata sobre el problema clásico de la comunicación privada, en el que Alice tiene un mensaje que quiere transmitir a Bob, manteniendo al mismo tiempo el contenido del mensaje oculto de una espía, Eva[1]. Pronto descubrirás que la criptografía abarca mucho más que la comunicación privada, pero este es el lugar lógico para comenzar.

4.1. ¿Qué es (y qué no es) la criptografía?

"Definir es limitar." —Oscar Wilde

La criptografía no es un hechizo mágico que resuelve todos los problemas de seguridad. Puede ofrecer soluciones a problemas claramente definidos, que a menudo abstraen preocupaciones importantes pero desordenadas del mundo real. La criptografía puede brindar garantías sobre lo que sucede en presencia de ciertas clases de ataques bien definidos. Sin embargo, estas garantías pueden no aplicarse si los atacantes del mundo real "no siguen las reglas" de un modelo de seguridad criptográfica.

Siempre ten esto en mente mientras definimos (es decir, limitamos) los problemas que resolveremos en este curso.

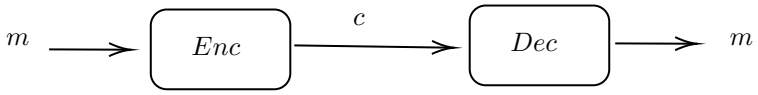
4.1.1. Conceptos básicos y terminología de cifrado

Comencemos a formalizar nuestro escenario que involucra a Alice, Bob y Eva. Alice tiene un mensaje m que desea enviar (de manera privada) a Bob. Llamamos m al *texto plano* (*plaintext*). Supondremos que

[1] *Eavesdropper* se refiere a alguien que escucha en secreto una conversación entre otros. El término se originó como una referencia a alguien que literalmente se colgaba de los aleros de un edificio para escuchar conversaciones dentro.

Alice transformará de alguna manera ese texto plano en un valor c (llamado *texto cifrado* o *ciphertext*), que enviará a Bob. El proceso de transformar m en c se denomina *cifrado* (*encryption*), y usaremos Enc para referirnos al algoritmo de cifrado. Cuando Bob recibe c, ejecuta un algoritmo de descifrado correspondiente, Dec, para recuperar el texto plano original m.

Supondremos que el texto cifrado puede ser observado por la espía Eva, por lo que el objetivo (informal) es que el texto cifrado sea comprensible para Bob pero incomprensible para Eva.



4.1.2. Secretos y el Principio de Kerckhoffs

Falta algo importante en este escenario. Si queremos que Bob pueda descifrar c, pero que Eva no pueda hacerlo, entonces Bob debe tener alguna información que Eva no tenga (¿ves por qué?). Algo debe mantenerse en secreto para Eva.

Podrías sugerir que los detalles de los algoritmos Enc y Dec sean secretos. Esto es como se hacía la criptografía durante la mayor parte de los últimos 2000 años, pero tiene grandes desventajas. Si el atacante eventualmente descubre los detalles de Enc y Dec, entonces la única forma de recuperar la seguridad es inventar nuevos algoritmos. Si tienes un sistema con muchos usuarios, la única forma de evitar que todos puedan leer los mensajes de los demás es inventar nuevos algoritmos para cada par de usuarios. ¡Inventar incluso un solo buen método de cifrado ya es lo suficientemente difícil!

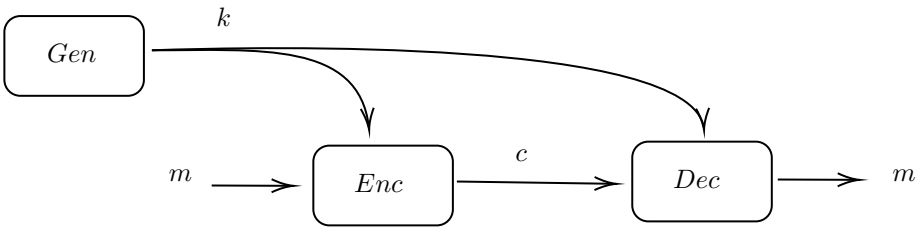
La primera persona en articular este problema fue Auguste Kerckhoffs. En 1883, formuló un conjunto de principios de diseño criptográfico. El ítem número 2 de su lista se conoce ahora como el **principio de Kerckhoffs**:

> **Principio de Kerckhoffs:** "Il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvenient tomber entre les mains de l'ennemi."
>
> **Traducción literal:** [El método] no debe depender de ser secreto y debe poder caer en manos del enemigo sin causar inconvenientes.
>
> **Conclusión:** Diseña tu sistema para que sea seguro incluso si el atacante tiene un conocimiento completo de todos sus algoritmos.

Si los algoritmos en sí mismos no son secretos, entonces debe haber otra información secreta en el sistema. Esa información se llama la *clave secreta*. La clave es simplemente una pieza adicional de información que se proporciona tanto al algoritmo Enc como al Dec. Otra forma de interpretar el principio de Kerckhoffs es que toda la seguridad del sistema debe concentrarse en el secreto de la clave, no en el secreto de los algoritmos. Si una clave secreta es comprometida, solo necesitas elegir una nueva, no reinventar un algoritmo de cifrado completamente nuevo. Múltiples usuarios pueden usar de forma segura el mismo algoritmo de cifrado, pero con claves secretas elegidas de forma independiente.

El proceso de elegir una clave secreta se llama *generación de claves*, y escribimos Gen para referirnos al algoritmo (aleatorio) de generación de claves. Llamamos al conjunto de tres algoritmos $(\text{Enc}, \text{Dec}, \text{Gen})$

un *esquema de cifrado*. Recuerda que el principio de Kerckhoffs dice que debemos asumir que un atacante conoce los detalles del algoritmo Gen. Pero también recuerda que conocer los detalles (es decir, el código fuente) de un algoritmo aleatorio no significa que conozcas la salida específica que dio cuando se ejecutó.



4.1.3. Excusas, excusas

Practiquemos algo de humildad. A continuación, se presenta una lista parcial de cuestiones que son claramente importantes para el problema de la comunicación privada, pero que no son abordadas por nuestra definición del problema.

- No estamos tratando de ocultar el *hecho* de que Alice le está enviando algo a Bob, solo queremos ocultar el *contenido* de ese mensaje. Ocultar la existencia de un canal de comunicación se denomina **esteganografía**.
- No consideraremos cómo el valor c llega de manera confiable de Alice a Bob. Simplemente daremos por sentado este aspecto.
- Por ahora, asumimos que Eva observa pasivamente la comunicación entre Alice y Bob. No estamos considerando un atacante que manipule c (haciendo que Bob reciba y descifre un valor diferente), aunque analizaremos estos ataques más adelante.
- No discutiremos cómo Alice y Bob obtienen realmente una clave secreta común en el mundo real. Este problema, conocido como **distribución de claves**, es claramente de suma importancia, y más adelante en curso trataremos algunos enfoques ingeniosos para resolverlo.

  El problema que estamos resolviendo ya es bastante complejo: una vez que dos usuarios han establecido una clave secreta compartida, ¿cómo pueden usar esa clave para comunicarse de forma privada?
- No abordaremos cómo Alice y Bob mantienen en secreto su clave, incluso después de haberla establecido. Una de mis descripciones favoritas de la criptografía es de Lea Kissner (antigua ingeniera principal de seguridad en Google): *"La criptografía es una herramienta para convertir muchos problemas diferentes en problemas de gestión de claves."*
- A lo largo de este curso, simplemente asumimos que los usuarios tienen la capacidad de muestrear cadenas aleatorias de forma uniforme. De hecho, sin aleatoriedad, no hay criptografía. En el mundo real, obtener bits verdaderamente aleatorios de computadoras deterministas es un desafío extremadamente complicado. John von Neumann dijo famosamente: *"Cualquiera que considere métodos aritméticos para producir dígitos aleatorios está, por supuesto, en estado de pecado."* Incluso cuando asumimos aleatoriedad uniforme, seguimos enfrentándonos a la cuestión no trivial de cómo usar esa aleatoriedad para la comunicación privada (y otras aplicaciones), además de cómo usar únicamente una cantidad manejable de aleatoriedad.

4.1.4. No criptografía

La gente utiliza muchas técnicas para intentar ocultar información, pero muchas de ellas son "no criptográficasza que no siguen el principio de Kerckhoffs:

- Métodos de codificación/decodificación como `base64`...

  joy of cryptography ↔ `b25sesBuZXJXYjcyB3awxsIHJ1YWQGhpwcw==`

  ... son útiles para incorporar datos binarios arbitrarios en un formato de archivo estructurado que admite tipos de caracteres limitados. Pero dado que la codificación/decodificación `base64` no implica información secreta, no añade nada en términos de seguridad.

- A veces, la forma más sencilla de describir un esquema de cifrado es mediante operaciones con cadenas binarias (es decir, datos de 0s y 1s). Como veremos, el *one-time pad* se define en términos de textos planos representados como cadenas de bits. (Esquemas futuros requerirán que las entradas se representen como una cadena de bits de una longitud específica, o como un elemento de $\mathbb{Z}_n$, etc.)

  Para comprender algunos algoritmos de este curso, puede ser necesario pensar en los datos convertidos a representación binaria. Al igual que con `base64`, representar las cosas en binario no tiene ningún efecto sobre la seguridad, ya que no implica ninguna información secreta. ¡Escribir algo en binario no es una medida de seguridad!

4.2. Especificidades del One-time pad

La gente ha intentado enviar mensajes secretos durante aproximadamente 2000 años, pero en realidad solo hay 2 ideas útiles anteriores a 1900 que tienen alguna relevancia para la criptografía moderna. La primera idea es el principio de Kerckhoffs, que ya has visto. La otra idea es el one-time pad (OTP), que ilustra varios conceptos importantes, e incluso se puede encontrar oculto en muchos esquemas de cifrado modernos.

El one-time pad a veces se llama “cifrado de Vernam” en honor a Gilbert Vernam, un ingeniero de telégrafos que patentó el esquema en 1919. Sin embargo, una descripción anterior del one-time pad fue descubierta recientemente en un texto de 1882 de Frank Miller sobre el cifrado telegráfico.

En la mayor parte de este curso, las claves secretas serán cadenas de bits. Generalmente usamos la variable λ para referirnos a la longitud (número de bits) de la clave secreta en un esquema, de modo que las claves son elementos del conjunto $\{0,1\}^\lambda$. En el caso del one-time pad, la elección de λ no afecta a la seguridad ($\lambda = 10$ es “igual de seguro” que $\lambda = 1000$); sin embargo, la longitud de las claves y los textos planos deben ser compatibles. En capítulos futuros, aumentar λ tiene el efecto de hacer que el esquema sea más difícil de romper. Por esa razón, λ a menudo se llama el **parámetro de seguridad** del esquema.

En el one-time pad, no solo las claves son cadenas de λ bits, sino que los textos planos y los textos cifrados también lo son. Debes considerar esto como una simple coincidencia, porque pronto encontraremos esquemas en los que las claves, los textos planos y los textos cifrados son cadenas de diferentes tamaños.

Los algoritmos específicos Gen, Enc y Dec para el one-time pad se dan a continuación:

OTP: \\ One-time pad		
Gen(λ):	Enc($k, m \in \{0,1\}^\lambda$):	Dec($k, c \in \{0,1\}^\lambda$):
$k \leftarrow \{0,1\}^\lambda$	return $k \oplus m$	return $k \oplus c$
return k		

Recuerda que "$k \leftarrow \{0,1\}^\lambda$" significa tomar una muestra de k uniformemente del conjunto de cadenas de λ bits. Esta elección uniforme de la clave es la única aleatoriedad en todos los algoritmos del one-time pad. Como veremos, toda su seguridad se deriva de esta elección de usar la distribución uniforme; las claves que se eligen de manera diferente no proporcionan una seguridad equivalente.

Ejemplo 4.1. Cifrar el siguiente texto plano m de 20 bits con la clave k de 20 bits usando el one-time pad resulta en el siguiente texto cifrado c:

$$\begin{array}{cll} & 11101111101111100011 & (m) \\ \oplus & 00011001110000111101 & (k) \\ \hline & 11110110011111011110 & (c = \mathrm{Enc}(k, m)) \end{array}$$

Descifrar el siguiente texto cifrado c usando la clave k resulta en el siguiente texto plano m:

$$\begin{array}{cll} & 00001001011110010000 & (c) \\ \oplus & 10010011101011100010 & (k) \\ \hline & 10011010110101110010 & (m = \mathrm{Dec}(k, c)) \end{array}$$

Nótese que Enc y Dec son esencialmente el mismo algoritmo (devuelven el XOR de los dos argumentos). Esto resulta en un pequeño nivel de conveniencia y simetría al implementar one-time pad, pero es más una coincidencia que algo verdaderamente fundamental sobre el cifrado. Más adelante veremos esquemas de cifrado cuyos algoritmos de cifrado y descifrado son muy diferentes.

4.2.1. Corrección

La primera propiedad del one-time pad que debemos confirmar es que el receptor realmente recupera el texto plano previsto al descifrar el texto cifrado. Sin esta propiedad, la idea de usar one-time pad para la comunicación parece absurda. Escrito matemáticamente:

Afirmación 4.2.1. *Para todo $k, m \in \{0,1\}^\lambda$, se cumple que $Dec(k, Enc(k, m)) = m$.*

Demostración. Esto se deduce al sustituir las definiciones de Enc y Dec de OTP, y luego aplicar las propiedades de XOR enumeradas en la Sección 1.3. Para todo $k, m \in \{0,1\}^\lambda$, tenemos:

$$\begin{aligned} \mathrm{Dec}(k, \mathrm{Enc}(k, m)) &= \mathrm{Dec}(k, k \oplus m) \\ &= k \oplus (k \oplus m) \\ &= (k \oplus k) \oplus m \\ &= 0^\lambda \oplus m \\ &= m. \end{aligned}$$

■

Ejemplo 4.2. Cifrar el siguiente texto plano m con la clave k resulta en el texto cifrado c, como sigue:

$$\begin{array}{cll} & 00110100110110001111 & (m) \\ \oplus & 11101010011010001101 & (k) \\ \hline & 11011110101100000010 & (c) \end{array}$$

Descifrar c usando la misma clave k resulta en el m original:

$$\begin{array}{cll} & 11011110101100000010 & (c) \\ \oplus & 11101010011010001101 & (k) \\ \hline & 00110100110110001111 & (m) \end{array}$$

4.2.2. Seguridad

Supongamos que Alice y Bob están usando one-time pad, pero les preocupa que un atacante vea su texto cifrado. No pueden presuponer lo que hará un atacante después de ver el texto cifrado. Pero les gustaría decir algo como: "debido a la forma específica en que se generó el texto cifrado, no revela ninguna información sobre el texto plano al atacante, independientemente de lo que el atacante haga con el texto cifrado".

Primero debemos especificar con precisión cómo se genera el texto cifrado. El algoritmo Enc ya describe el proceso, pero está escrito desde el punto de vista de Alice y Bob. Cuando hablamos de seguridad, tenemos que pensar en lo que hacen Alice y Bob, ¡pero desde el punto de vista del atacante! Desde el punto de vista de Eva, Alice usa una clave que se eligió de una manera específica (uniformemente al azar), cifra un texto plano con esa clave usando OTP y, finalmente, revela solo el texto cifrado resultante (y no la clave) a Eva.

Más formalmente, desde la perspectiva de Eva, ver un texto cifrado corresponde a recibir una salida del siguiente algoritmo:

EAVESDROP($m \in \{0,1\}^\lambda$):
$k \leftarrow \{0,1\}^\lambda$
$c := k \oplus m$
return c

Es fundamental entender qué representa el algoritmo EAVESDROP. Su propósito **no es describir lo que hace el atacante**, sino el proceso (¡realizado por Alice y Bob!) que genera lo que **el atacante puede observar** (el valor del cifrado c). Siempre modelamos al atacante como un proceso (no especificado) que recibe la salida de este algoritmo EAVESDROP. Nuestro objetivo es establecer afirmaciones como: "la salida de EAVESDROP no revela la entrada m".

Es importante destacar que Eva tiene pleno conocimiento de todos los detalles del diseño del algoritmo EAVESDROP. Sin embargo, una vez que se entrega el mensaje, Eva ya no tiene acceso a los valores seleccionados aleatoriamente (en este caso, la clave k). En otras palabras, Eva sabe que se ha escogido una clave k de forma aleatoria y que se ha calculado el XOR entre el mensaje m y la clave k para producir el texto cifrado c. No obstante, hasta que no observa el valor de c, Eva no tiene información sobre el valor específico de k elegido en esta ejecución del algoritmo.

Este enfoque resalta que EAVESDROP actúa como una caja negra que impide al atacante observar los valores escogidos aleatoriamente, incluso teniendo pleno conocimiento del funcionamiento interno del algoritmo. La única forma en que el atacante podría conocer estos valores aleatorios sería que el diseño del algoritmo permita que esta información se filtre, de manera explícita o implícita, en el texto cifrado que el atacante observa como salida del algoritmo.

EAVESDROP es un *algoritmo aleatorizado*: recuerda que "$k \leftarrow \{0,1\}^\lambda$" significa tomar una muestra de k de la distribución uniforme sobre cadenas de λ bits. Si llamas a EAVESDROP varias veces, incluso con la misma entrada, es probable que obtengas salidas diferentes. En lugar de pensar en "EAVESDROP(m)" como una sola cadena, debes pensar en ello como una *distribución de probabilidad* sobre cadenas. Cada vez que llamas a EAVESDROP(m), ves una *muestra* de esa distribución.

Ejemplo 4.3. Consideremos $\lambda = 3$ y calculemos manualmente las distribuciones EAVESDROP(010) y EAVESDROP(111). En cada caso, EAVESDROP elige un valor de k uniformemente en $\{0,1\}^3$ — cada uno de los posibles valores con probabilidad $1/8$. Para cada elección posible de k, podemos calcular cuál será la salida de EAVESDROP (c):

EAVESDROP(010):

Pr	k	**salida** $c = k \oplus 010$
$\frac{1}{8}$	000	010
$\frac{1}{8}$	001	011
$\frac{1}{8}$	010	000
$\frac{1}{8}$	011	001
$\frac{1}{8}$	100	110
$\frac{1}{8}$	101	111
$\frac{1}{8}$	110	100
$\frac{1}{8}$	111	101

EAVESDROP(111):

Pr	k	**salida** $c = k \oplus 111$
$\frac{1}{8}$	000	111
$\frac{1}{8}$	001	110
$\frac{1}{8}$	010	101
$\frac{1}{8}$	011	100
$\frac{1}{8}$	100	011
$\frac{1}{8}$	101	010
$\frac{1}{8}$	110	001
$\frac{1}{8}$	111	000

Por lo tanto, la distribución EAVESDROP(010) asigna probabilidad $1/8$ a 010, probabilidad $1/8$ a 011, y así sucesivamente.

En este ejemplo, observe cómo cada cadena en $\{0,1\}^3$ aparece exactamente una vez en la columna c de EAVESDROP(010). Esto significa que EAVESDROP asigna una probabilidad de $1/8$ a cada cadena en $\{0,1\}^3$, lo cual es otra forma de decir que la distribución es la distribución uniforme en $\{0,1\}^3$. Lo mismo se puede decir sobre la distribución EAVESDROP(111), también. ¡Ambas distribuciones son simplemente la distribución uniforme disfrazada!

No hay nada especial en 010 o 111 en el ejemplo anterior. Para cualquier λ y cualquier $m \in \{0,1\}^\lambda$, la distribución EAVESDROP(m) es la distribución uniforme sobre $\{0,1\}^\lambda$.

Afirmación 4.2.2. *Para cada $m \in \{0,1\}^\lambda$, la distribución EAVESDROP(m) es la **distribución uniforme** sobre $\{0,1\}^\lambda$. Por lo tanto, para todo m, $m' \in \{0,1\}^\lambda$, las distribuciones EAVESDROP(m) y EAVESDROP(m') son idénticas.*

Demostración. Fijemos arbitrariamente $m, c \in \{0,1\}^\lambda$. Calcularemos la probabilidad de que EAVESDROP(m) produzca la salida c. Ese evento ocurre solo cuando

$$c = k \oplus m \iff k = m \oplus c.$$

La equivalencia se deduce de las propiedades de XOR dadas en la Sección 1.3. Es decir,

$$\Pr[\text{EAVESDROP}(m) = c] = \Pr[k = m \oplus c],$$

donde la probabilidad se calcula sobre la elección uniforme de

$$k \leftarrow \{0,1\}^\lambda.$$

Estamos considerando una elección específica para m y c, por lo que solo hay un valor de k que hace que $k = m \oplus c$ sea verdadero (hace que m se cifre a c), y ese valor es exactamente $m \oplus c$. Dado que k se elige uniformemente de $\{0,1\}^\lambda$, la probabilidad de elegir el valor particular $k = m \oplus c$ es $1/2^\lambda$.

En resumen, para cada m y c, la probabilidad de que EAVESDROP(m) produzca c es exactamente $1/2^\lambda$. Esto significa que la salida de EAVESDROP(m), para cualquier m, sigue la distribución uniforme. ■

Una forma de interpretar esta afirmación de seguridad en términos más prácticos:

Si un atacante ve un único valor de c cifrado con el one-time pad, donde la clave se elige uniformemente y se mantiene en secreto para el atacante, entonces el texto cifrado aparece distribuido uniformemente.

¿Por qué es esto significativo? Desde el punto de vista del atacante, supongamos que alguien elige un texto plano m y tú ves el texto cifrado resultante: una muestra de la distribución EAVESDROP(m). ¡Pero esta es una distribución de la que tú mismo puedes obtener muestras, incluso si no conoces m! Podrías haber elegido un m' totalmente diferente y ejecutar EAVESDROP(m') en tu imaginación, y esto habría producido la misma distribución que EAVESDROP(m).

El texto cifrado "real" que ves no contiene ninguna información sobre m si es posible obtener muestras de la misma distribución sin siquiera conocer m.

4.2.3. Discusión

- ¿No hay una paradoja? La Afirmación 4.2.1 dice que c siempre puede ser descifrado para obtener m, ¡pero la Afirmación 4.2.2 dice que c no contiene información sobre m! La respuesta a este enigma es que la Afirmación 4.2.1 habla sobre lo que se puede hacer con el conocimiento de la clave k (la perspectiva de Alice y Bob). La Afirmación 4.2.2 habla sobre la distribución de salida del algoritmo EAVESDROP, que no incluye a k (la perspectiva de Eva). En resumen, si conoces k, entonces puedes descifrar c para obtener m; si no conoces k, entonces c no contiene información sobre m (de hecho, parece distribuido uniformemente). Esto se debe a que m, c, k están correlacionados de una manera delicada.[2]
- ¿No hay otra paradoja? La Afirmación 4.2.2 dice que la salida de EAVESDROP(m) no depende de m, ¡pero podemos ver el algoritmo EAVESDROP usando literalmente su argumento m justo ahí en la última línea! La respuesta a este enigma quizás se ilustra mejor con las ilustraciones anteriores de las distribuciones EAVESDROP(010) y EAVESDROP(111). Las dos tablas de valores son de hecho diferentes (por lo que la elección de $m \in \{010, 111\}$ claramente tiene algún efecto), pero representan la misma distribución de probabilidad (ya que el orden no importa y las dos tablas difieren solo en el orden de los valores). La Afirmación 4.2.2 considera solo la distribución de probabilidad resultante.

[2]Se profundizará más en la naturaleza de esta correlación más adelante.

- Probablemente pienses en la seguridad en términos de un "objetivo" concreto para el atacante: recuperar la clave, recuperar el texto plano, etc. La Afirmación 4.2.2 realmente no se refiere a los atacantes de esa manera, y ciertamente no especifica un objetivo. Más bien, estamos pensando en la seguridad comparando con un mundo "ideal" hipotético. ¡Me daría por satisfecho si el atacante solo ve una fuente de bits uniformes, porque en este mundo hipotético no hay claves ni textos planos que recuperar! La Afirmación 4.2.2 dice que cuando realmente usamos el one-time pad, se ve justo como este mundo hipotético, desde el punto de vista del atacante. Si imaginamos algún "objetivo" para el atacante en este tipo de razonamiento, es detectar que los textos cifrados no siguen una distribución uniforme. Al mostrar que el atacante ni siquiera puede lograr este modesto objetivo, se demuestra que el atacante posiblemente no podría lograr otros objetivos más naturales, como la recuperación de claves y la recuperación de texto plano.

4.2.4. Limitaciones

El one-time pad es increíblemente limitado en la práctica. Principalmente:

- Sus claves son tan largas como los textos planos que cifran. Esto es básicamente inevitable y conduce a una especie de dilema del huevo y la gallina en la práctica: Si dos usuarios quieren transmitir de forma privada un mensaje de λ bits, primero necesitan acordar de forma privada una cadena de λ bits.
- Una clave solo se puede usar para cifrar un texto plano (de ahí, one-time pad: cifrado de un sólo uso). De hecho, podemos ver que la subrutina EAVESDROP en la Afirmación 4.2.2 no proporciona ninguna forma para que una persona que la llama garantice que dos textos planos se cifren con la misma clave, por lo que no está claro cómo usar la Afirmación 4.2.2 para argumentar sobre lo que sucede en el one-time pad cuando las claves se reutilizan intencionalmente de esta manera.

A pesar de estas limitaciones, el one-time pad ilustra ideas fundamentales que aparecen en la mayoría de las formas de cifrado en este curso.

Capítulo 5

Fundamentos de seguridad demostrable

Edgar Allan Poe no solo fue un autor, sino también un entusiasta de la criptografía. Una vez escribió, en una discusión sobre el estado del arte en criptografía:[1]

> "La ingenuidad humana no puede inventar un cifrado que la ingenuidad humana no pueda resolver."

Esta fue una evaluación precisa de la criptografía que existía en 1841. Siempre que alguien ideaba un método de cifrado, otra persona inevitablemente encontraba una manera de romperlo, y el juego del gato y el ratón se repetía una y otra vez.

La criptografía moderna del siglo XXI, sin embargo, es diferente. Este curso te presentará muchos esquemas cuya seguridad podemos **demostrar** en un sentido muy específico. Los creadores de códigos pueden ganar contra los descifradores.

Solo es posible **demostrar** cosas sobre seguridad teniendo definiciones formales de lo que significa ser "seguro". Este capítulo trata sobre las habilidades fundamentales que giran en torno a las definiciones de seguridad: cómo escribirlas, cómo entenderlas e interpretarlas, cómo demostrar la seguridad utilizando la **técnica híbrida**, y cómo demostrar la inseguridad utilizando ataques contra la definición de seguridad.

5.1. Cómo escribir una definición de seguridad

Hasta ahora, la única forma de criptografía que hemos visto es el one-time pad, por lo que nuestra discusión sobre seguridad ha sido bastante específica para el one-time pad. Sería preferible tener un vocabu-

[1] Edgar Allan Poe, "A Few Words on Secret Writing," Graham's Magazine, Julio 1841, v19.

lario para hablar sobre seguridad en un sentido más general, de modo que podamos preguntar si **cualquier** esquema de cifrado es seguro.

En esta sección, desarrollaremos dos definiciones de seguridad para el cifrado.

5.1.1. Lo que no forma parte de una definición de seguridad

Una definición de seguridad debe dar garantías sobre lo que le puede suceder a un sistema en presencia de un atacante. Pero no todas las propiedades importantes de un sistema se refieren a un atacante. Para el cifrado específicamente:

- No hacemos referencia a ningún atacante cuando decimos que el algoritmo Enc toma dos argumentos (una clave y un texto plano), o que el algoritmo Gen no toma ningún argumento. Especificar los tipos de entradas/salidas (es decir, la "firma de la función") de los diversos algoritmos, por lo tanto, no es una declaración sobre seguridad. Llamamos a estas propiedades la **sintaxis** del esquema.
- Incluso si no hay un atacante, sigue siendo importante que el descifrado sea una inversa del cifrado. Esta no es una propiedad de seguridad del esquema de cifrado. En cambio, la llamamos una propiedad de **corrección**.

Definición 5.1.1 (Sintaxis del cifrado de clave simétrica (SKE))**.** *Un esquema de cifrado de clave simétrica (SKE) consta de los siguientes algoritmos:*

- *Gen: un algoritmo aleatorizado que produce una clave $k \in \mathcal{K}$.*
- *Enc: un algoritmo (posiblemente aleatorizado) que toma una clave $k \in \mathcal{K}$ y un texto plano $m \in \mathcal{M}$ como entrada, y produce un texto cifrado $c \in \mathcal{C}$.*
- *Dec: un algoritmo determinista que toma una clave $k \in \mathcal{K}$ y un texto cifrado $c \in \mathcal{C}$ como entrada, y produce un texto plano $m \in \mathcal{M}$.*

Llamamos $\mathcal{K}$ el espacio de claves, $\mathcal{M}$ el espacio de mensajes, y $\mathcal{C}$ el espacio de textos cifrados del esquema. A veces nos referimos a todo el esquema (la colección de los tres algoritmos) con una sola variable Σ. Cuando lo hacemos, escribimos $\Sigma.Gen$, $\Sigma.Enc$, $\Sigma.Dec$, $\Sigma.\mathcal{K}$, $\Sigma.\mathcal{M}$, y $\Sigma.\mathcal{C}$ para referirnos a sus componentes.

Definición 5.1.2 (Corrección de SKE)**.** *Un esquema de cifrado Σ satisface la **corrección** si para toda $k \in \Sigma.\mathcal{K}$ y toda $m \in \Sigma.\mathcal{M}$,*

$$\Pr[\Sigma.Dec(k, \Sigma.Enc(k, m)) = m] = 1.$$

La definición está escrita en términos de una probabilidad porque se permite que Enc sea un algoritmo aleatorizado. En otras palabras, descifrar un texto cifrado con la misma clave que se usó para el cifrado siempre debe dar como resultado el texto plano original.

Ejemplo 5.1 (Ejemplo de sintaxis sin corrección)**.** Un esquema de cifrado puede tener la sintaxis apropiada pero aun así tener un comportamiento degenerado como $\text{Enc}(k, m) = 0$ (es decir, cada texto plano se "cifra" a 0). Tal esquema no satisfaría la propiedad de corrección.

Ejemplo 5.2 (Ejemplo de corrección sin seguridad)**.** Un esquema diferente definido por $\text{Enc}(k, m) = m$ (es decir, el "texto cifrado" siempre es igual al propio texto plano) y $\text{Dec}(k, c) = c$ sí satisface la propiedad de corrección, pero no satisfaría ninguna propiedad de seguridad razonable.

5.2. Definición de seguridad al estilo "Real-vs-Aleatorio"

Intentemos crear una definición de seguridad que formalice la siguiente idea intuitiva:

> "un esquema de cifrado es bueno si sus textos cifrados le parecen basura aleatoria a un atacante."

Las definiciones de seguridad siempre consideran la perspectiva del **atacante** del sistema. ¿Cuál es la "interfaz" que Alice y Bob exponen al atacante mediante su uso de la criptografía, y esa interfaz en particular beneficia al atacante?

En este ejemplo, estamos considerando un escenario donde el atacante puede observar textos cifrados. ¿Cómo se generan exactamente estos textos cifrados? ¿Cuáles son las entradas para Enc (clave y texto plano), y cómo se eligen?

Clave: Es difícil imaginar cualquier tipo de seguridad útil si el atacante conoce la clave. Por lo tanto, consideramos que la clave se mantiene en secreto para el atacante. Por supuesto, la clave se genera de acuerdo con el algoritmo Gen del esquema.

En este punto del curso, consideramos esquemas de cifrado donde la clave se usa para cifrar solo un texto plano. De alguna manera, esta restricción debe capturarse en nuestra definición de seguridad. Más adelante, consideraremos definiciones de seguridad que consideran una clave que se usa para cifrar muchas cosas.

Texto plano: Resulta útil considerar que el atacante en realidad elige los textos planos. Esta es una elección "pesimista", ya que le da mucho poder al atacante. Sin embargo, si el esquema de cifrado es realmente seguro cuando el atacante elige los textos planos, entonces también es seguro en escenarios más realistas donde el atacante tiene cierta incertidumbre sobre los textos planos.

Estas aclaraciones nos permiten completar más detalles sobre nuestra idea informal de seguridad:

> "un esquema de cifrado es bueno si sus textos cifrados le parecen basura aleatoria a un atacante... cuando cada clave es secreta y se usa para cifrar solo un texto plano, incluso cuando el atacante elige los textos planos."

Una forma concisa de expresar todos estos detalles es considerar al atacante como un **programa que llama** a la siguiente subrutina:

$\underline{\text{CTXT}(m \in \Sigma.\mathcal{M}):}$
$k \leftarrow \Sigma.\text{Gen}$
$c := \Sigma.\text{Enc}(k, m)$
return c

Un programa que llama puede elegir el argumento para la subrutina (en este caso, un texto plano) y ver solo el valor de retorno resultante (en este caso, un texto cifrado). El programa que llama **no puede** ver los valores de las variables de ámbito privado (como k en este caso). Si el programa que llama realiza muchas llamadas a la subrutina, se elige una nueva clave k cada vez.

La interacción entre un atacante (programa que llama) y esta subrutina CTXT parece capturar el escenario relevante. Nos gustaría decir que las salidas de la subrutina CTXT se distribuyen uniformemente. Una forma conveniente de expresar esta propiedad es decir que esta subrutina CTXT debería tener el mismo efecto en cada programa que llama que una subrutina CTXT que (explícitamente) muestrea su salida uniformemente.

$\underline{\text{CTXT}(m \in \Sigma.\mathcal{M})}$:
$k \leftarrow \Sigma.\text{Gen}$
$c := \Sigma.\text{Enc}(k, m)$
return c

vs.

$\underline{\text{CTXT}(m \in \Sigma.\mathcal{M})}$:
$c \leftarrow \Sigma.\mathcal{C}$
return c

Intuitivamente, ningún programa que llama debería tener ninguna forma de determinar cuál de estas dos implementaciones está respondiendo a las llamadas de la subrutina.

En resumen, podemos definir la seguridad para el cifrado de la siguiente manera:

> Un esquema de cifrado es bueno si, cuando se introducen sus algoritmos Gen y Enc en la plantilla de la subrutina CTXT anterior, las dos implementaciones de CTXT inducen un comportamiento idéntico en cada programa que llama.

En unas pocas páginas, introduciremos notación formal y definiciones para los conceptos introducidos aquí. En particular, tanto el programa que llama como la subrutina pueden ser algoritmos aleatorizados, por lo que debemos tener cuidado con lo que queremos decir con "comportamiento idéntico".

Ejemplo 5.3. El cifrado de Vernam (one-time pad) se define con Gen muestreando k uniformemente de $\{0,1\}^\lambda$ y $\text{Enc}(k, m) = k \oplus m$. Satisface nuestra nueva propiedad de seguridad ya que, cuando introducimos estos algoritmos en la plantilla anterior, obtenemos las siguientes dos implementaciones de la subrutina:

$\underline{\text{CTXT}(m)}$:
$k \leftarrow \{0,1\}^\lambda$ \\ Gen de OTP
$c := k \oplus m$ \\ Enc de OTP
return c

vs.

$\underline{\text{CTXT}(m)}$:
$c \leftarrow \{0,1\}^\lambda$ \\ Muestreo de $\mathcal{C}$ de OTP
return c

Y estas dos implementaciones tienen el mismo efecto en todos los programas que llaman.

5.2.1. Definición de seguridad al estilo "Intercambiabilidad"

Aquí hay una idea intuitiva diferente de seguridad:

> Un esquema de cifrado es bueno si los cifrados de m_0 se parecen a los cifrados de m_1 para un atacante (para todos los posibles m_0, m_1).

Como antes, estamos considerando un escenario donde el atacante ve algunos textos cifrados. Estos textos cifrados se generan con alguna clave; ¿de dónde viene esa clave? Estos textos cifrados cifran ya sea algún m_0 o algún m_1; ¿de dónde vienen m_0 y m_1? Podemos responder estas preguntas de manera similar al ejemplo anterior. Los textos planos m_0 y m_1 pueden ser elegidos por el atacante. La clave se elige de acuerdo con Gen para que permanezca secreta para el atacante (y se usa para generar solo un texto cifrado).

> Un esquema de cifrado es bueno si los cifrados de m_0 se parecen a los cifrados de m_1 para un atacante, cuando cada clave es secreta y se usa para cifrar solo un texto plano, incluso cuando el atacante elige m_0 y m_0.

Como antes, formalizamos esta idea imaginando al atacante como un programa que llama a una interfaz particular. Esta vez, el atacante elegirá dos textos planos m_0 y m_1 y recibirá un texto cifrado a cambio.[2] Dependiendo de si m_0 o m_1 se cifra realmente, esas interfaces se implementan de la siguiente manera:

EAVESDROP$(m_0, m_1 \in \Sigma.\mathcal{M})$:
$k \leftarrow \Sigma.\text{Gen}$
$c := \Sigma.\text{Enc}(k, m_0)$
return c

;

EAVESDROP$(m_0, m_1 \in \Sigma.\mathcal{M})$:
$k \leftarrow \Sigma.\text{Gen}$
$c := \Sigma.\text{Enc}(k, m_1)$
return c

Ahora, la manera formal de decir que los cifrados de m_0 "se parecen" a los cifrados de m_1 es:

> Un esquema de cifrado es bueno si, cuando se introducen sus algoritmos Gen y Enc en la plantilla de las subrutinas EAVESDROP anteriores, las dos implementaciones de EAVESDROP inducen un comportamiento idéntico en cada programa que llama.

Ejemplo 5.4. ¿Satisface el cifrado de Vernam (one-time pad) esta nueva propiedad de seguridad? Para averiguarlo, introducimos sus algoritmos en la plantilla anterior y obtenemos las siguientes implementaciones:

EAVESDROP(m_0, m_1):
$k \leftarrow \{0,1\}^\lambda$ \\ Gen de OTP
$c := k \oplus m_0$ \\ Enc de OTP
return c

;

EAVESDROP(m_0, m_1):
$k \leftarrow \{0,1\}^\lambda$ \\ Gen de OTP
$c := k \oplus m_1$ \\ Enc de OTP
return c

Si estas dos implementaciones tienen el mismo efecto en todos los programas que las llaman (y de hecho lo tienen), entonces diríamos que OTP satisface esta propiedad de seguridad.

¿Es esta una mejor/peor forma de definir la seguridad que la forma anterior? Una definición de seguridad considera un atacante cuyo objetivo es distinguir textos cifrados reales de valores aleatorios (paradigma real contra aleatorio), y la otra considera un atacante cuyo objetivo es distinguir textos cifrados reales de dos textos planos diferentes (paradigma de intercambiabilidad). ¿Es una "correcta" y la otra "incorrecta"? Guardamos tal discusión para más adelante en el capítulo.

5.3. Formalismos para Definiciones de Seguridad

Hasta ahora, hemos definido la seguridad en términos de una única subrutina autocontenida, y hemos imaginado al atacante como un programa que llama a esta subrutina. Más adelante en el curso necesitaremos generalizar más allá de una sola subrutina, a una colección de subrutinas que comparten información de estado (privada). Siguiendo con la terminología de software, llamamos a esta colección una **librería**:

[2] Puede haber otras formas razonables de formalizar esta idea intuitiva de seguridad. Por ejemplo, podríamos optar por darle al atacante dos textos cifrados en lugar de uno, y exigir que el atacante no pueda determinar cuál de ellos cifra m_0 y cuál cifra m_1.

Definición 5.3.1 (Librerías)**.** *Una **librería** $\mathcal{L}$ es una colección de subrutinas y variables privadas/estáticas. La **interfaz** de una librería consiste en los nombres, tipos de argumentos y tipo de salida de todas sus subrutinas (al igual que una interfaz de Java). Si un programa $\mathcal{A}$ incluye llamadas a subrutinas en la interfaz de $\mathcal{L}$, entonces escribimos $\mathcal{A} \diamond \mathcal{L}$ para denotar el resultado de **enlazar** $\mathcal{A}$ con $\mathcal{L}$ de la manera natural (respondiendo a esas llamadas de subrutinas usando la implementación especificada en $\mathcal{L}$). Escribimos $\mathcal{A} \diamond \mathcal{L} \Rightarrow z$ para denotar el evento de que el programa $\mathcal{A} \diamond \mathcal{L}$ produce el valor z.*

Si $\mathcal{A}$ o $\mathcal{L}$ es un programa que realiza elecciones aleatorias, entonces la salida de $\mathcal{A} \diamond \mathcal{L}$ es una variable aleatoria. A menudo es útil considerar probabilidades como $\Pr[\mathcal{A} \diamond \mathcal{L} \Rightarrow \mathit{true}]$.

Ejemplo 5.5. Aquí hay una librería familiar:

$\mathcal{L}$

$\underline{\text{CTXT}(m)\text{:}}$
$k \leftarrow \{0,1\}^\lambda$
$c := k \oplus m$
return c

Y aquí hay un posible programa que la llama:

$\mathcal{A}$

$m \leftarrow \{0,1\}^\lambda$
$c := \text{CTXT}(m)$
return $m \stackrel{?}{=} c$

Esperamos que puedas convencerte de que

$$\Pr[\mathcal{A} \diamond \mathcal{L} \Rightarrow \text{true}] = 1/2^\lambda.$$

Si este $\mathcal{A}$ está enlazado a una librería diferente, su probabilidad de salida puede ser diferente. Si un programa diferente se enlaza a esta $\mathcal{L}$, la probabilidad de salida puede ser diferente.

Ejemplo 5.6. Una librería puede contener varias subrutinas y variables privadas que se mantienen estáticas entre llamadas a subrutinas. Por ejemplo, aquí hay una librería simple que elige una cadena s uniformemente y permite que el programa que llama adivine s.

$\mathcal{L}$

$s \leftarrow \{0,1\}^\lambda$

$\underline{\text{RESET}()\text{:}}$
$s \leftarrow \{0,1\}^\lambda$

$\underline{\text{GUESS}(x \in \{0,1\}^\lambda)\text{:}}$
return $x \stackrel{?}{=} s$

Nuestra convención es que el código fuera de una subrutina (como la primera línea aquí) se ejecuta una vez en el momento de la inicialización. Las variables definidas en el momento de la inicialización (como s aquí) están disponibles en todos los ámbitos de las subrutinas (pero no para el programa que llama).

5.3.1. Librerías intercambiables

La idea de que "ningún programa que llama se comporta de manera diferente en presencia de dos librerías" sigue teniendo sentido incluso para librerías con varias subrutinas. Dado que este es un concepto tan común, le dedicamos una nueva definición:

Definición 5.3.2 (Librerías intercambiables)**.** *Sean $\mathcal{L}_0$ y $\mathcal{L}_1$ dos librerías que tienen la misma interfaz. Decimos que $\mathcal{L}_0$ y $\mathcal{L}_1$ son intercambiables, y lo escribimos $\mathcal{L}_0 \equiv \mathcal{L}_1$, si para todos los programas $\mathcal{A}$ que producen un valor booleano,*

$$\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow \mathit{true}] = \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow \mathit{true}].$$

Esta definición considera programas que dan una salida booleana. Imagine un programa / atacante cuyo único objetivo es distinguir dos librerías particulares (de hecho, a menudo nos referimos al programa como un **distinguidor**). Una salida booleana es suficiente para esa tarea. Se puede pensar en el bit de salida como la "conjetura" del programa sobre a qué librería cree que está enlazado.

La distinción entre "el programa da verdadero como salida" y "el programa da falso como salida" no es significativa. Si dos librerías no afectan la probabilidad de que el programa de verdadero como salida, entonces tampoco afectan su probabilidad de dar falso como salida:

$$\begin{aligned} & \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow \text{true}] = \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow \text{true}] \\ \Leftrightarrow \quad & 1 - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow \text{true}] = 1 - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow \text{true}] \\ \Leftrightarrow \quad & \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{left}} \Rightarrow \text{false}] = \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{right}} \Rightarrow \text{false}]. \end{aligned}$$

Ejemplo 5.7. Aquí hay algunas formas muy simples y directas en que dos librerías pueden ser intercambiables. Debes tener claro que cada par de librerías tiene un comportamiento idéntico y, por lo tanto, un efecto idéntico en todos los programas que llaman.

A pesar de ser ejemplos muy simples, cada uno de estos conceptos aparece como un **bloque de fundamental** en una demostración de seguridad real en este curso.

- **Ejemplo 1:**

```
FOO(x):                          ≡   FOO(x):
if x es par then                     if x es par then
    return 0                             return 0
else if x es impar then              else if x es impar then
    return 1                             return 1
else                                 else
    return -1                            return ∞
```

Su única diferencia ocurre en un bloque de código inalcanzable.

- **Ejemplo 2:**

```
FOO(x):                          ≡   FOO(x):
return BAR(x, x)                     return BAR(x, 0^λ)

BAR(a, b):                           BAR(a, b):
k ← {0,1}^λ                          k ← {0,1}^λ
return k ⊕ a                         return k ⊕ a
```

Su única diferencia es el valor que asignan a una variable que nunca se utiliza en realidad.

- **Ejemplo 3:**

FOO(x, n): for $i := 1$ to λ do BAR(x, i)	$\equiv$	FOO(x, n): for $i := 1$ to n do BAR(x, i) for $i := n+1$ to λ do BAR(x, i)	Su única diferencia es que una librería desdobla un bucle que ocurre en la otra librería.

- **Ejemplo 4:**

FOO(x): $k \leftarrow \{0,1\}^\lambda$ $y \leftarrow \{0,1\}^\lambda$ return $k \oplus$ $y \oplus x$	$\equiv$	FOO(x): $k \leftarrow \{0,1\}^\lambda$ return $k \oplus$ BAR(x) BAR(x): $y \leftarrow \{0,1\}^\lambda$ return $y \oplus x$	Su única diferencia es que una librería incluye (inline) una llamada a una subrutina que ocurre en la otra librería.

Ejemplo 5.8. Aquí hay más ejemplos sencillos de librerías intercambiables que involucran aleatoriedad.

- **Ejemplo 5:**

FOO(): $x \leftarrow \{0,1\}^\lambda$ $y \leftarrow \{0,1\}$ return $x \| y$	$\equiv$	FOO(): $z \leftarrow \{0,1\}^{2\lambda}$ return z	La distribución uniforme sobre cadenas actúa independientemente sobre diferentes caracteres en la cadena ("$\|\|$" denota concatenación).

- **Ejemplo 6:**

$k \leftarrow \{0,1\}^\lambda$ FOO(x): return $k \oplus x$	$\equiv$	FOO(x): if k no esta definida: $k \leftarrow \{0,1\}^\lambda$ return $k \oplus x$	Muestreo de un valor "ávidamente" (tan pronto como sea posible) frente a muestreo de un valor "perezosamente" (en el último momento posible antes de que se necesite el valor). Asumimos que k es estático/global a través de muchas llamadas a FOO, e inicialmente indefinido.

5.4. Reformulaciones formales de conceptos anteriores

Ahora podemos reformular nuestras definiciones de seguridad de la sección anterior de manera formal.

La definición de seguridad en el paradigma "real-vs-aleatorio" para un cifrado se puede expresar de la siguiente manera:

Definición 5.4.1 (Cifrados uniformes de un solo uso)**.** *Un esquema de cifrado* Σ *tiene* ***cifrados uniformes de un solo uso*** *si:*

$$\boxed{\begin{array}{l}\mathcal{L}^{\Sigma}_{otu}\\ \underline{CTXT(m \in \Sigma.\mathcal{M}):}\\ k \leftarrow \Sigma.Gen\\ c := \Sigma.Enc(k, m)\\ return\ \ c\end{array}} \quad \equiv \quad \boxed{\begin{array}{l}\mathcal{L}^{\Sigma}_{otu\text{-}\$}\\ \underline{CTXT(m \in \Sigma.\mathcal{M}):}\\ c \leftarrow \Sigma.\mathcal{C}\\ return\ \ c\end{array}}$$

Equivalentemente, un esquema de cifrado Σ tiene textos cifrados uniformes de un solo uso si para todos los programas $\mathcal{A}$ que producen un valor booleano,

$$\Pr[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{otu} \Rightarrow true] = \Pr[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{otu\text{-}\$} \Rightarrow true].$$

En otras palabras, si completas los detalles de Σ (es decir, el comportamiento de su Gen y Enc) en estas dos "plantillas" de librería, y obtienes dos librerías que son intercambiables (es decir, tienen el mismo efecto en todos los programas que las llaman), diremos que Σ tiene textos cifrados uniformes de un solo uso.

A lo largo de este curso, utilizaremos el símbolo "$" para denotar aleatoriedad (como en las librerías de la definición anterior).[3]

La definición de seguridad en el paradigma "intercambiabilidad" se puede expresar de la siguiente manera:

Definición 5.4.2 (Intercambiabilidad de un solo uso)**.** *Un esquema de cifrado Σ tiene* ***intercambiabilidad de un solo uso*** *si:*

$$\boxed{\begin{array}{l}\mathcal{L}^{\Sigma}_{otint\text{-}0}\\ \underline{EAVESDROP(m_0, m_1 \in \Sigma.\mathcal{M}):}\\ k \leftarrow \Sigma.Gen\\ c := \Sigma.Enc(k, m_0)\\ return\ \ c\end{array}} \quad \equiv \quad \boxed{\begin{array}{l}\mathcal{L}^{\Sigma}_{otint\text{-}1}\\ \underline{EAVESDROP(m_0, m_1 \in \Sigma.\mathcal{M}):}\\ k \leftarrow \Sigma.Gen\\ c := \Sigma.Enc(k, m_1)\\ return\ \ c\end{array}}$$

Profundicemos en la idea presentada en la definición anterior. Representaremos la interacción de un atacante $\mathcal{A}$ con las librerías $\mathcal{L}^{\Sigma}_{\text{otint-0}}$ y $\mathcal{L}^{\Sigma}_{\text{otint-1}}$ como un juego. En este juego, se selecciona aleatoriamente una de las dos librerías, y esta se conecta con $\mathcal{A}$ sin que el atacante sepa cuál fue elegida. El objetivo de la definición es capturar la noción de intercambiabilidad entre las librerías: estas se consideran intercambiables si la probabilidad de que $\mathcal{A}$ logre distinguir con cuál está interactuando no mejore la probabilidad de acertar al azar, es decir, de simplemente adivinar.

Definimos un juego $\text{Priv}^{\text{EAV}}_{\mathcal{A},\Sigma}$, basado en $\mathcal{A}$ y Σ, de la siguiente manera:

[3] Es bastante común en la literatura de Ciencias de la Computación usar el símbolo "$" cuando se hace referencia a la aleatoriedad. Esto proviene de la idea de pensar en los algoritmos aleatorizados como algoritmos que "lanzan monedas". Por lo tanto, los algoritmos aleatorizados necesitan tener cambio (es decir, dinero) disponible. Por convención, la aleatoriedad viene en dólares estadounidenses.

El juego de intercambiabilidad $\text{Priv}^{\text{EAV}}_{\mathcal{A},\Sigma}$:

1. El atacante $\mathcal{A}$ produce un par de mensajes $m_0, m_1 \in \Sigma.\mathcal{M}$.
2. Se elige un bit de manera aleatoria $b \in \{0,1\}$ y se enlaza $\mathcal{A}$ con $\mathcal{L}^{\Sigma}_{\text{otint-}b}$, es decir, $\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{otint-}b}$.
3. A produce un bit b', es decir, $\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{otint-}b} \Rightarrow b'$.
4. La salida del juego se define como 1 si $b' = b$, y 0 en caso contrario. Escribimos $\text{Priv}^{\text{EAV}}_{\mathcal{A},\Sigma} = 1$ si la salida del juego es 1 y en este caso decimos que $\mathcal{A}$ tiene éxito.

Como se señaló anteriormente, es trivial para $\mathcal{A}$ tener éxito con una probabilidad de $1/2$ al producir una conjetura aleatoria. La intercambiabilidad entre las librerías $\mathcal{L}^{\Sigma}_{\text{otint-0}}$ y $\mathcal{L}^{\Sigma}_{\text{otint-1}}$ requiere que sea imposible para cualquier $\mathcal{A}$ hacerlo mejor que con una probabilidad de $1/2$. Así que:

Un esquema de cifrado Σ tiene **intercambiabilidad de un solo uso** si:

$$\Pr[\text{Priv}^{\text{EAV}}_{\mathcal{A},\Sigma} = 1] = \frac{1}{2}.$$

Anteriormente, en la Afirmación 4.2.2 argumentamos que los textos cifrados del one-time pad (OTP) siguen la distribución uniforme. Esto en realidad muestra que OTP satisface la definición de cifrados uniformes de un solo uso.

Afirmación 5.4.3. *El one-time pad (OTP) satisface la propiedad de cifrados uniformes de un solo uso. En otras palabras:*

$\mathcal{L}^{OTP}_{otu}$	$\equiv$	$\mathcal{L}^{OTP}_{otu\text{-}\$}$
$\underline{CTXT(m \in \{0,1\}^\lambda):}$ $k \leftarrow \{0,1\}^\lambda$ \\ *OTP.Gen* *return* $k \oplus m$ \\ *OTP.Enc*(k,m)	$\equiv$	$\underline{CTXT(m \in \{0,1\}^\lambda):}$ $c \leftarrow \{0,1\}^\lambda$ \\ *OTP.C* *return* c

5.4.1. Discusión, Errores comunes

Es un error común imaginar que el programa $\mathcal{A}$ está vinculado *simultáneamente* a ambas librerías, pero esto no es lo que dice la definición. La definición de $\mathcal{L}_0 \equiv \mathcal{L}_1$ se refiere a dos ejecuciones diferentes: una donde $\mathcal{A}$ está vinculado solo a $\mathcal{L}_0$ durante toda su vida útil, y otra donde $\mathcal{A}$ está vinculado solo a $\mathcal{L}_1$ durante toda su vida útil. Nunca hay un momento en el que algunas de las llamadas a subrutinas de $\mathcal{A}$ sean respondidas por $\mathcal{L}_0$ y otras por $\mathcal{L}_1$. Esta es una distinción especialmente importante cuando $\mathcal{A}$ realiza varias llamadas a subrutinas en una sola ejecución.

Otro error común es la confusión sobre la diferencia entre los algoritmos de un esquema de cifrado (p. ej., lo que se muestra en la descripción del OTP en la Sección 4.2) y las librerías utilizadas en una definición de seguridad (p. ej., lo que se muestra en la Definición 5.4.2). La gran diferencia es:

- Los algoritmos del esquema muestran el punto de vista de un usuario. Por ejemplo, el algoritmo Enc toma dos entradas: una clave y un texto plano. ¿Existe alguna forma de describir un algoritmo que toma dos argumentos que no sea escribiendo algo como la descripción del OTP en la Sección 4.2?
- Las librerías capturan el punto de vista del atacante de un escenario particular, donde los usuarios utilizan los algoritmos criptográficos de una manera muy específica. Por ejemplo, cuando hablamos de seguridad del cifrado, ¡no garantizamos la seguridad cuando Alice le permite al atacante elegir su clave de cifrado! Pero permitir que el atacante elija el texto plano está bien; podemos garantizar la seguridad en ese escenario. Es por eso que la Definición 5.4.1 describe una subrutina que llama a Enc sobre un texto plano que es elegido por el atacante, pero sobre una clave k elegida por la librería.

Una definición de seguridad dice que alguna tarea (p. ej., distinguir entre textos cifrados y basura aleatoria) es imposible, cuando al atacante se le permite cierta influencia sobre las entradas de los algoritmos (p. ej., elección completa de los textos planos, pero sin influencia sobre la clave), y se le permite ver ciertas salidas de esos algoritmos (p. ej., textos cifrados).

Es **incorrecto** resumir la definición de intercambiabilidad de un solo uso como: "No se me permite elegir qué cifrar, tengo que pedirle al atacante que elija por mí". La interpretación correcta es: "Si cifro solo un texto plano por clave, entonces es seguro cifrar textos incluso si el atacante ve el texto cifrado resultante e incluso si tiene alguna influencia o información parcial sobre lo que estoy cifrando, porque esta es la situación capturada en la librería de intercambiabilidad de un solo uso".

5.4.2. El Principio de Kerckhoffs, Revisado

El Principio de Kerckhoffs dice que se debe asumir que el atacante tiene un conocimiento completo de los algoritmos que se utilizan y que la elección de las claves es lo único desconocido para el atacante. Veamos cómo se refleja el Principio de Kerckhoffs en nuestras definiciones de seguridad formales.

Supongamos que yo escribo el código fuente de dos librerías, y tu objetivo es escribir un distinguirdor efectivo. Así que tú estudias el código fuente de las dos librerías y escribes el mejor distinguirdor que existe. Sería certero decir que tu distinguirdor "sabe" qué algoritmos se utilizan en las librerías, porque fue diseñado basándose en el código fuente de estas librerías. La definición de intercambiabilidad considera literalmente todos los programas, por lo que también debe considerar distinguidores como el tuyo que "saben" qué algoritmos se están utilizando.

Sin embargo, hay una distinción importante que hacer. Si tú sabes que podrías estar vinculado a una librería que ejecuta la sentencia "$k \leftarrow \{0,1\}^{\lambda}$", eso no significa que conozcas el valor real de k que se eligió en el tiempo de ejecución. Nuestra convención es que todas las variables dentro de la librería tienen un ámbito privado, y el distinguidor solo puede aprender sobre ellas indirectamente a través de las salidas de las subrutinas. ¡En el juego de intercambiabilidad de librerías, no se te permite elegir un distinguidor diferente basándote en las elecciones aleatorias que hace la librería! Después de que construimos un distinguidor, medimos su efectividad en términos de probabilidades que tienen en cuenta todos los resultados posibles de las elecciones aleatorias en el sistema.

En resumen, el distinguidor "sabe" qué algoritmos se están utilizando (¡y cómo se están utilizando!) porque la construcción del distinguidor puede depender de las 2 librerías específicas que consideramos.

El distinguidor "no sabe" cosas como las claves secretas porque la construcción del distinguidor no puede depender del resultado del muestreo aleatorio realizado en tiempo de ejecución.

El Principio de Kerckhoffs, aplicado a nuestra terminología formal:
Asuma que el atacante conoce todos los hechos del universo, excepto por:

1. a cuál de las dos posibles librerías está vinculado en cualquier ejecución particular, y
2. las elecciones aleatorias que hará la librería durante cualquier ejecución particular (que generalmente se asignan a variables de ámbito privado dentro de la librería).

5.5. Cómo demostrar inseguridad con ataques

Siempre definimos la seguridad con respecto a dos librerías — o, si lo prefieres, dos plantillas de librería que describen cómo insertar los algoritmos de un esquema criptográfico en dos librerías. Si las dos librerías que se obtienen (después de completar los detalles de un esquema particular) son intercambiables, entonces decimos que el esquema satisface la propiedad de seguridad. Si queremos mostrar que algún esquema es **inseguro**, solo tenemos que demostrar que existe un programa que se comporta de manera diferente en presencia de esas dos librerías.

Ilustraremos este proceso con el siguiente esquema de cifrado, que es como el one-time pad pero que usa AND bit a bit en lugar de XOR:

OTP modificado

$\mathcal{K} = \{0,1\}^\lambda$
$\mathcal{M} = \{0,1\}^\lambda$
$\mathcal{C} = \{0,1\}^\lambda$

Gen:
$k \leftarrow \{0,1\}^\lambda$
return k

Enc(k, m):
return $k \& m$ \\ AND bit a bit

No he mostrado el algoritmo Dec, porque de hecho no hay forma de escribir uno que satisfaga el requisito de corrección. Pero pretendamos que aún no nos hemos dado cuenta de eso y preguntémonos si este esquema de cifrado satisface las dos propiedades de seguridad definidas anteriormente.

Afirmación 5.5.1. *El OTP modificado* ***no*** *tiene cifrados uniformes de un solo uso (Definición 5.4.1).*

Demostración. Para ver si el OTP modificado tiene textos cifrados uniformes de un solo uso, tenemos que insertar sus algoritmos en las dos librerías de la Definición 5.4.1 y ver si las librerías resultantes son intercambiables. Estamos considerando las siguientes dos librerías:

$\mathcal{L}^\Sigma_{\text{otu}}$

CTXT$(m \in \{0,1\}^\lambda)$:
$k \leftarrow \{0,1\}^\lambda$ \\ Σ.Gen
$c := k \& m$ \\ Σ.Enc
return c

$\mathcal{L}^\Sigma_{\text{otu-\$}}$

CTXT$(m \in \{0,1\}^\lambda)$:
$c \leftarrow \{0,1\}^\lambda$ \\ $\Sigma.\mathcal{C}$
return c

Para mostrar que estas dos librerías **no** son intercambiables, necesitamos construir un distinguidor que se comporte de manera diferente al vincularse con cada librería. El programa que llama debe realizar una o más llamadas a la subrutina CTXT. Eso significa que necesita elegir la entrada m, y debe sacar alguna conclusión (sobre a cuál de las dos librerías está vinculado) basándose en el valor de la salida que obtiene. ¿Qué m debería elegir el programa que llama como entrada para CTXT? ¿Qué debería buscar el programa que llama en los valores de salida?

Hay muchas maneras válidas de escribir un buen distinguidor, y tal vez tú puedas pensar en varias. Un buen enfoque es observar que el AND bit a bit con k nunca puede "convertir un 0 en un 1". Así que tal vez el programa que llama debería elegir que m consista en todos 0s. Cuando $m = 0^\lambda$, la librería $\mathcal{L}^\Sigma_{\text{otu}}$ siempre devolverá todos ceros, pero la librería $\mathcal{L}^\Sigma_{\text{otu-\$}}$ puede devolver cadenas con tanto 0s como 1s.

Podemos formalizar esta idea con el siguiente programa que llama:

$\mathcal{A}$
$c := \text{CTXT}(0^\lambda)$
return $c \stackrel{?}{=} 0^\lambda$

A continuación, es obligatorio demostrar que este distinguidor se comporta de manera diferente cuando se vincula a cada una de estas dos librerías (no basta solo con escribirlo y decir que hace lo que queremos que haga).

$\mathcal{A}$
$c := \text{CTXT}(0^\lambda)$
return $c \stackrel{?}{=} 0^\lambda$

$\diamond$

$\mathcal{L}^\Sigma_{\text{otu}}$

$\underline{\text{CTXT}(m \in \{0,1\}^\lambda):}$
$k \leftarrow \{0,1\}^\lambda$
$c := k \& m$
return c

Cuando $\mathcal{A}$ está vinculado a $\mathcal{L}^\Sigma_{\text{otu}}$, c se calcula como $k \& 0^\lambda$. No importa cuál sea k, el resultado siempre es $c = 0^\lambda$. Por lo tanto, $\mathcal{A}$ siempre devolverá verdadero. En otras palabras, $\Pr[\mathcal{A} \diamond \mathcal{L}^\Sigma_{\text{otu}} \Rightarrow \text{true}] = 1$.

$\mathcal{A}$
$c := \text{CTXT}(0^\lambda)$
return $c \stackrel{?}{=} 0^\lambda$

$\diamond$

$\mathcal{L}^\Sigma_{\text{otu-\$}}$

$\underline{\text{CTXT}(m \in \{0,1\}^\lambda):}$
$c \leftarrow \{0,1\}^\lambda$
return c

Cuando $\mathcal{A}$ está vinculado a $\mathcal{L}^\Sigma_{\text{otu-\$}}$, c se elige uniformemente de $\{0,1\}^\lambda$. La probabilidad de que c resulte ser todo ceros es $1/2^\lambda$. En otras palabras, $\Pr[\mathcal{A} \diamond \mathcal{L}^\Sigma_{\text{otu-\$}} \Rightarrow \text{true}] = 1/2^\lambda$.

Dado que estas dos probabilidades son diferentes, esto muestra que $\mathcal{L}^\Sigma_{\text{otu}} \not\equiv \mathcal{L}^\Sigma_{\text{otu-\$}}$. En otras palabras, el esquema no satisface la propiedad de textos cifrados uniformes.

■

De momento tenemos dos definiciones de seguridad. ¿Satisface este esquema de cifrado una pero no la otra?

Afirmación 5.5.2. *El OTP modificado no tiene intercambiabilidad de un solo uso (Definición 5.4.2).*

Demostración. Esta afirmación se refiere a una definición de seguridad diferente, que involucra dos librerías diferentes.

Cuando insertamos los detalles del OTP modificado en las librerías de la Definición 5.4.2, obtenemos lo siguiente:

$\mathcal{L}^{\Sigma}_{\text{otint-0}}$

EAVESDROP(m_0, m_1):
$k \leftarrow \{0,1\}^{\lambda}$ \\ Σ.Gen
$c := k \& m_0$ \\ Σ.Enc(k, m_0)
return c

$\mathcal{L}^{\Sigma}_{\text{otint-1}}$

EAVESDROP(m_0, m_1):
$k \leftarrow \{0,1\}^{\lambda}$ \\ Σ.Gen
$c := k \& m_1$ \\ Σ.Enc(k, m_1)
return c

Ahora necesitamos construir un programa que se comporte de manera diferente en presencia de estas dos librerías. Podemos usar la misma idea general que la última vez, pero no el mismo programa distinguidor, ya que estas librerías proporcionan una interfaz diferente. En este ejemplo, el programa necesita llamar a la subrutina EAVESDROP que toma dos argumentos m_0 y m_1. ¿Cómo debería el programa que llama elegir m_0 y m_1? ¿Qué dos textos planos tienen textos cifrados de aspecto diferente?

Un buen enfoque es elegir que m_0 sean todos ceros y m_1 sean todos unos. Sabemos de antes que un texto plano de todos ceros siempre se cifra en un texto cifrado de todos ceros, por lo que el programa que construimos puede verificar esa condición. Más formalmente, podemos construir el siguiente programa:

$\mathcal{A}$
$c :=$ EAVESDROP$(0^{\lambda}, 1^{\lambda})$
return $c \stackrel{?}{=} 0^{\lambda}$

A continuación, es obligatorio calcular sus probabilidades de salida en presencia de las dos librerías (no basta con escribir el programa y decir que hace lo que queremos que haga).

$\mathcal{A}$
$c :=$ EAVESDROP$(0^{\lambda}, 1^{\lambda})$
return $c \stackrel{?}{=} 0^{\lambda}$

$\diamond$

$\mathcal{L}^{\Sigma}_{\text{otint-0}}$

EAVESDROP(m_0, m_1):
$k \leftarrow \{0,1\}^{\lambda}$
$c := k \& m_0$
return c

Cuando $\mathcal{A}$ está vinculado a $\mathcal{L}^{\Sigma}_{\text{otint-0}}$, c se calcula como un cifrado de $m_0 = 0^{\lambda}$. No importa cuál sea k, el resultado siempre es todo ceros. Por lo tanto, $\Pr[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{otint-0}} \Rightarrow \text{true}] = 1$.

$\mathcal{A}$
$c :=$ EAVESDROP$(0^{\lambda}, 1^{\lambda})$
return $c \stackrel{?}{=} 0^{\lambda}$

$\diamond$

$\mathcal{L}^{\Sigma}_{\text{otint-1}}$

EAVESDROP(m_0, m_1):
$k \leftarrow \{0,1\}^{\lambda}$
$c := k \& m_1$
return c

Cuando $\mathcal{A}$ está vinculado a $\mathcal{L}^{\Sigma}_{\text{otint-1}}$, c se calcula como un cifrado de $m_1 = 1^{\lambda}$. En otras palabras, $c := k \& 1^{\lambda}$. Pero el AND bit a bit de cualquier cadena k con todos 1s es simplemente k. Así que c es simplemente igual a k, que se eligió uniformemente al azar. La probabilidad de que un c uniformemente aleatorio resulte ser todo ceros es $\Pr[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{otint-1}} \Rightarrow \text{true}] = 1/2^{\lambda}$.

Dado que estas dos probabilidades son diferentes, $\mathcal{L}^{\Sigma}_{\text{otint-0}} \not\equiv \mathcal{L}^{\Sigma}_{\text{otint-1}}$ y el esquema no tiene intercambiabilidad de un solo uso. ■

5.6. Cómo demostrar seguridad

Demostramos que el one-time pad satisface la propiedad de textos cifrados uniformes (Afirmación 4.2.2) calculando cuidadosamente ciertas probabilidades. Esta no será una estrategia sostenible a medida que las cosas se compliquen más adelante en el curso. En esta sección introduciremos una técnica para probar propiedades de seguridad, que generalmente evita cálculos de probabilidad tediosos.

5.6.1. Encadenando varios componentes

Antes de llegar a una prueba de seguridad, introducimos un lema muy útil. Considere un programa compuesto como $\mathcal{A} \diamond \mathcal{L}_1 \diamond \mathcal{L}_2$. Nuestra convención es que las llamadas a subrutinas solo ocurren de izquierda a derecha a través del símbolo $\diamond$, por lo que en este ejemplo, $\mathcal{L}_1$ puede realizar llamadas a subrutinas en $\mathcal{L}_2$, pero no al revés. Dependiendo del contexto, a veces puede ser conveniente interpretar $\mathcal{A} \diamond \mathcal{L}_1 \diamond \mathcal{L}_2$ como:

- $(\mathcal{A} \diamond \mathcal{L}_1) \diamond \mathcal{L}_2$: una composición programa/librería vinculado a $\mathcal{L}_2$. Después de todo, $\mathcal{A} \diamond \mathcal{L}_1$ es un programa que realiza llamadas a la interfaz de $\mathcal{L}_2$.
- o $\mathcal{A} \diamond (\mathcal{L}_1 \diamond \mathcal{L}_2)$: $\mathcal{A}$ vinculado a una librería compuesta. Después de todo, $\mathcal{A}$ es un programa que realiza llamadas a la interfaz de $(\mathcal{L}_1 \diamond \mathcal{L}_2)$.

La colocación de los paréntesis no afecta la funcionalidad del programa general, al igual que la forma en que dividir un programa real en diferentes archivos fuente no afecta su funcionalidad.

De hecho, cada prueba de seguridad en este libro tendrá algunos pasos intermedios que involucran librerías compuestas. Haremos un uso intensivo del siguiente resultado útil:

Lema 5.6.1. *Si $\mathcal{L}_0 \equiv \mathcal{L}_1$ entonces, para cualquier librería $\mathcal{L}^*$, tenemos:*

$$\mathcal{L}^* \diamond \mathcal{L}_0 \equiv \mathcal{L}^* \diamond \mathcal{L}_1.$$

Demostración. Observe que estamos comparando $\mathcal{L}^* \diamond \mathcal{L}_0$ y $\mathcal{L}^* \diamond \mathcal{L}_1$ como librerías compuestas. Por lo tanto, consideramos un programa $\mathcal{A}$ que está vinculado a $\mathcal{L}^* \diamond \mathcal{L}_0$ o $\mathcal{L}^* \diamond \mathcal{L}_1$.

Sea $\mathcal{A}$ un programa arbitrario. Debemos mostrar que $\mathcal{A} \diamond (\mathcal{L}^* \diamond \mathcal{L}_0)$ y $\mathcal{A} \diamond (\mathcal{L}^* \diamond \mathcal{L}_1)$ tienen distribuciones de salida idénticas. Como se mencionó anteriormente, podemos interpretar $\mathcal{A} \diamond \mathcal{L}^* \diamond \mathcal{L}_0$ como un programa de llamada $\mathcal{A}$ vinculado a la librería $\mathcal{L}^* \diamond \mathcal{L}_0$, pero también como un programa de llamada $\mathcal{A} \diamond \mathcal{L}^*$ vinculado a la librería $\mathcal{L}_0$. Dado que $\mathcal{L}_0 \equiv \mathcal{L}_1$, intercambiar $\mathcal{L}_0$ por $\mathcal{L}_1$ no tiene ningún efecto en la salida de ningún programa de llamada. En particular, no tiene ningún efecto cuando el programa de llamada resulta ser el programa compuesto $\mathcal{A} \diamond \mathcal{L}^*$. Por lo tanto, tenemos:

$$\begin{aligned}
\Pr[\mathcal{A} \diamond (\mathcal{L}^* \diamond \mathcal{L}_0) \Rightarrow \text{true}] &= \Pr[(\mathcal{A} \diamond \mathcal{L}^*) \diamond \mathcal{L}_0 \Rightarrow \text{true}] && \text{(cambio de perspectiva)} \\
&= \Pr[(\mathcal{A} \diamond \mathcal{L}^*) \diamond \mathcal{L}_1 \Rightarrow \text{true}] && \text{(dado que } \mathcal{L}_0 \equiv \mathcal{L}_1\text{)} \\
&= \Pr[\mathcal{A} \diamond (\mathcal{L}^* \diamond \mathcal{L}_1) \Rightarrow \text{true}] && \text{(cambio de perspectiva)}
\end{aligned}$$

Dado que $\mathcal{A}$ era arbitrario, hemos demostrado el lema. ■

5.6.2. Un ejemplo de demostración

En esta sección probaremos algo sobre el siguiente esquema, que cifra dos veces con OTP, usando claves independientes:

OTP doble			
	Gen:	Enc$((k_1,k_2),m)$:	Dec$((k_1,k_2),c)$:
$\mathcal{K} = \{0,1\}^\lambda \times \{0,1\}^\lambda$	$k_1 \leftarrow \{0,1\}^\lambda$	$c_1 := k_1 \oplus m$	$c_1 := k_2 \oplus c$
$\mathcal{M} = \{0,1\}^\lambda$	$k_2 \leftarrow \{0,1\}^\lambda$	$c := k_2 \oplus c_1$	$m := k_1 \oplus c_1$
$\mathcal{C} = \{0,1\}^\lambda$	return (k_1,k_2)	return c	return m

No sería demasiado difícil mostrar directamente que los textos cifrados en este esquema se distribuyen uniformemente, como lo hicimos para OTP simple. Sin embargo, la técnica de demostración que hemos descrito en esta sección nos permitirá aprovechar lo que ya sabemos sobre OTP de una manera elegante y evitar cualquier cálculo de probabilidad.

Afirmación 5.6.2. *El OTP doble tiene cifrados uniformes de un sólo uso (Definición 5.4.1).*

Demostración. En términos de librerías, debemos demostrar que

$\mathcal{L}^{\Sigma}_{\text{otu}}$

CTXT$(m \in \{0,1\}^\lambda)$:
$k_1 \leftarrow \{0,1\}^\lambda$, $k_2 \leftarrow \{0,1\}^\lambda$ } Gen
$c_1 := k_1 \oplus m$, $c := k_2 \oplus c_1$ } Enc
return c

$\equiv$

$\mathcal{L}^{\Sigma}_{\text{otu-\$}}$

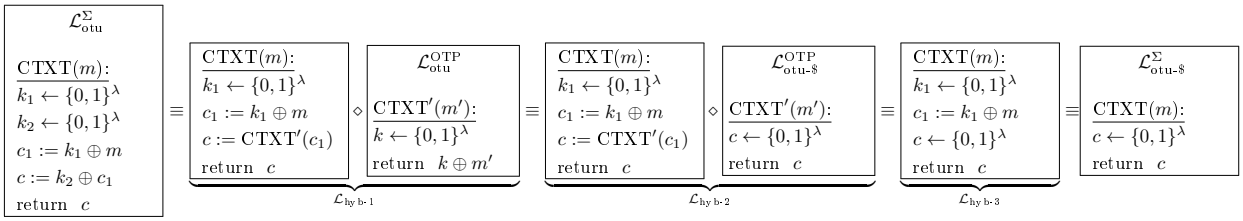
CTXT$(m \in \{0,1\}^\lambda)$:
$c \leftarrow \{0,1\}^\lambda$ \\ $\Sigma.\mathcal{C}$
return c

En lugar de comparar directamente estas dos librerías, introduciremos algunas librerías adicionales $\mathcal{L}_{\text{hyb-1}}$, $\mathcal{L}_{\text{hyb-2}}$, $\mathcal{L}_{\text{hyb-3}}$ y mostraremos que:

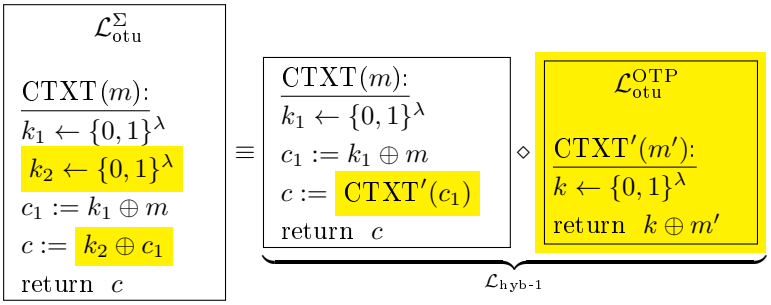
$$\mathcal{L}_{\text{otu}} \equiv \mathcal{L}_{\text{hyb-1}} \equiv \mathcal{L}_{\text{hyb-2}} \equiv \mathcal{L}_{\text{hyb-3}} \equiv \mathcal{L}_{\text{otu-\$}}$$

Dado que el símbolo $\equiv$ es transitivo, esto logrará nuestro objetivo.

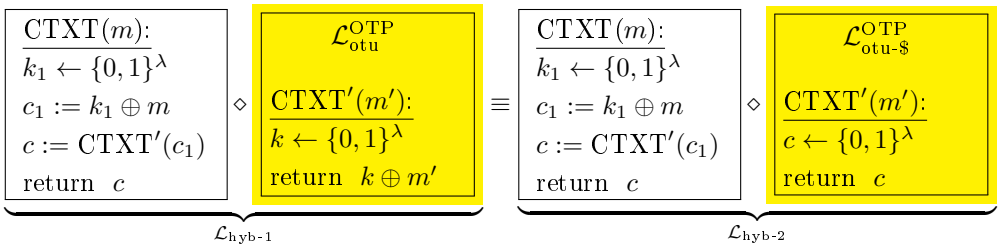
Las librerías intermedias se denominan *híbridas*, ya que contendrán una mezcla de características de los dos "puntos finales" de esta secuencia. Estas librerías híbridas se eligen de manera que sea muy fácil demostrar que las librerías consecutivas en esta secuencia son intercambiables. Las híbridas particulares que tenemos en mente aquí son:



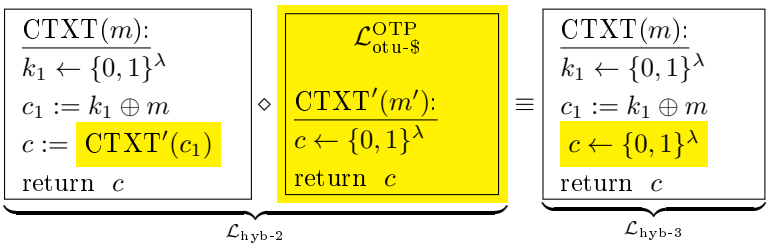
A continuación, proporcionamos una justificación para cada "$\equiv$" en la expresión anterior. Para cada par de librerías adyacentes, destacamos sus diferencias a continuación:



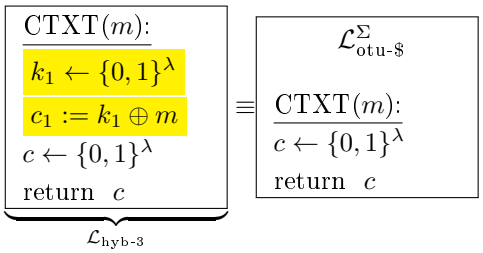
La única diferencia entre estas dos librerías es que las expresiones resaltadas se han factorizado en una subrutina separada y algunas variables han sido renombradas. En ambas librerías, c se calcula como el XOR de c_1 y una cadena elegida uniformemente. Estas diferencias no tienen ningún efecto en el comportamiento de cualquier programa vinculadas a las librerías. Es importante destacar que la subrutina que hemos factorizado es exactamente la misma que la de la librería $\mathcal{L}^{\text{OTP}}_{\text{otu}}$ (aparte de renombrar la subrutina).



La Afirmación 5.4.3 dice que $\mathcal{L}^{\text{OTP}}_{\text{otu}} \equiv \mathcal{L}^{\text{OTP}}_{\text{otu-\$}}$, por lo que el Lema 5.6.1 dice que podemos reemplazar una instancia de $\mathcal{L}^{\text{OTP}}_{\text{otu}}$ en una librería compuesta con $\mathcal{L}^{\text{OTP}}_{\text{otu-\$}}$, como hemos hecho aquí. Este cambio no tendrá ningún efecto en el comportamiento de cualquier programa vinculadas a las librerías.



La única diferencia entre estas dos librerías es que se ha insertado el código de una subrutina directamente en el lugar donde es llamada (*inlining*). Esta diferencia no tiene ningún efecto en el comportamiento de cualquier programa vinculadas a las librerías.



La única diferencia entre estas dos librerías es que se han eliminado las dos líneas resaltadas. Pero debería quedar claro que estas líneas no tienen ningún efecto: k_1 se usa solo para calcular c_1, que nunca se vuelve a usar. Por lo tanto, esta diferencia no tiene ningún efecto en el comportamiento de cualquier programa vinculadas a las librerías. La librería híbrida final es exactamente $\mathcal{L}_{\text{otu-\$}}$. Hemos demostrado que $\mathcal{L}_{\text{otu}} \equiv \mathcal{L}_{\text{otu-\$}}$, lo que significa que este esquema de cifrado tiene textos cifrados uniformes de un solo uso.

■

5.6.3. Resumen

Ya hemos visto nuestro primer ejemplo de prueba de seguridad. De manera resumida, lo que hemos hecho hasta ahora es:

- Demostrar la seguridad significa mostrar que dos librerías particulares, digamos $\mathcal{L}_0$ y $\mathcal{L}_1$, son intercambiables.
- A menudo, $\mathcal{L}_0$ y $\mathcal{L}_1$ son significativamente diferentes, lo que dificulta su comparación directa. Para que la comparación sea más manejable, podemos mostrar una secuencia de librerías híbridas, comenzando con $\mathcal{L}_0$ y terminando con $\mathcal{L}_1$. La idea es dividir la gran "brecha" entre $\mathcal{L}_0$ y $\mathcal{L}_1$ en otras más pequeñas que sean más fáciles de justificar.
- Es útil pensar en "comenzar" en $\mathcal{L}_0$ y luego realizar una secuencia de pequeñas modificaciones, con el objetivo de llegar finalmente a $\mathcal{L}_1$. Siempre se debe justificar por qué cada modificación no afecta el comportamiento de cualquier programa que se vincula con las librerías (es decir, por qué las dos librerías antes/después de su modificación son intercambiables).

- Como se discutió en la Sección 5.3.1, cosas simples como la inserción en línea/factorización de subrutinas, el cambio de variables no utilizadas, el cambio de sentencias inalcanzables o el desdoble de bucles son siempre modificaciones "permitidas" en una prueba, ya que no tienen ningún efecto en el comportamiento de cualquier programa que se vincula con las librerías.

5.6.4. Un Ejemplo contrastante

Generalmente, la frontera entre lo seguro y lo inseguro es muy delgada. Hagamos un pequeño cambio al esquema de cifrado anterior para ilustrar este punto. En lugar de aplicar OTP al texto plano dos veces, con claves independientes, ¿qué sucedería si usamos *la misma* clave?

OTP dOB$\ell\exists$			
	Gen:	Enc(k, m):	Dec(k, c):
$\mathcal{K} = \{0,1\}^\lambda$	$k \leftarrow \{0,1\}^\lambda$	$c_1 := k \oplus m$	$c_1 := k \oplus c$
$\mathcal{M} = \{0,1\}^\lambda$	return k	$c := k \oplus c_1$	$m := k \oplus c_1$
$\mathcal{C} = \{0,1\}^\lambda$		return c	return m

Probablemente notaste que el texto cifrado c se calcula como $c := k \oplus (k \oplus m)$, que es solo una forma elegante de decir $c := m$. Ciertamente, este tipo de "OTP dOB$\ell\exists$" no es seguro.

Con fines educativos, intentemos repetir los pasos de nuestra prueba de seguridad anterior en este esquema (inseguro) y veamos dónde **fallan las cosas**. Si quisiéramos mostrar que el OTP dOB$\ell\exists$ tiene textos cifrados uniformes, tendríamos que mostrar que las siguientes dos librerías son intercambiables:

$\mathcal{L}^{\Sigma}_{\text{otu}}$

CTXT$(m \in \{0,1\}^\lambda)$:
$k \leftarrow \{0,1\}^\lambda$ \\ Gen
$c_1 := k \oplus m$
$c := k \oplus c_1$ } Enc
return c

$\overset{?}{\equiv}$

$\mathcal{L}^{\Sigma}_{\text{otu-\$}}$

CTXT$(m \in \{0,1\}^\lambda)$:
$c \leftarrow \{0,1\}^\lambda$ \\ $\Sigma.\mathcal{C}$
return c

En la prueba anterior, el primer paso fue factorizar las sentencias "$k_2 \leftarrow \{0,1\}^\lambda$; $c := k_2 \oplus c_1$" en una subrutina separada, para que pudiéramos argumentar que el resultado de c estaba distribuido uniformemente. Si hacemos algo análogo con este ejemplo, obtenemos:

$\mathcal{L}^{\Sigma}_{\text{otu}}$

CTXT(m):
$k \leftarrow \{0,1\}^\lambda$
$c_1 := k \oplus m$
$c := k \oplus c_1$
return c

$\overset{?}{\equiv}$

CTXT(m):
$c_1 := k \oplus m$ \\ ??
$c :=$ CTXT$'(c_1)$
return c

$\diamond$

$\mathcal{L}^{\text{OTP}}_{\text{otu}}$

CTXT$'(m')$:
$k \leftarrow \{0,1\}^\lambda$
return $k \oplus m'$

$\mathcal{L}_{\text{hyb}}$

¿Ves el problema? En "$\mathcal{L}_{\text{hyb}}$", hemos intentado mover la variable k a $\mathcal{L}_{\text{otu}}^{\text{OTP}}$. Dado que ahora k es una variable privada de $\mathcal{L}_{\text{otu}}^{\text{OTP}}$, cada operación que queramos hacer con k tiene que ser proporcionada por su librería contenedora $\mathcal{L}_{\text{otu}}^{\text{OTP}}$. Pero hay una discrepancia: $\mathcal{L}_{\text{otu}}^{\text{OTP}}$ solo nos da una forma de usar k en una expresión XOR, mientras que necesitamos usar la misma k en dos expresiones XOR para que coincida con el comportamiento de $\mathcal{L}_{\text{otu}}^{\text{OTP}}$. La librería compuesta $\mathcal{L}_{\text{hyb}}$ tiene una referencia no resuelta a k en la línea "$c_1 := k \oplus m$", y por lo tanto no tiene el mismo comportamiento que $\mathcal{L}_{\text{otu}}^{\text{OTP}}$. [4] Este es el paso de la prueba de seguridad que falla.

Aquí hay una forma más conceptual de entender qué salió mal aquí. La propiedad importante del OTP es que sus textos cifrados parecen uniformes cuando la clave se usa para cifrar *solo un* texto plano. Esta variante de "OTP doble" usa OTP de una manera que no cumple con esa condición y, por lo tanto, no proporciona ninguna garantía de seguridad. La prueba (exitosa) anterior pudo factorizar algunos XOR en términos de $\mathcal{L}_{\text{otu}}^{\text{OTP}}$ sin romper nada, y así es como sabemos que el esquema estaba usando OTP de una manera que es consistente con su garantía de seguridad.

Como puedes ver, con suerte, el proceso de una prueba de seguridad proporciona una forma de detectar este tipo de problemas. Es muy común en una prueba de seguridad factorizar algunas sentencias en términos de una librería de alguna otra definición de seguridad. Este paso solo se puede realizar con éxito si la criptografía subyacente se está utilizando de una manera apropiada.

5.7. Cómo comparar/contrastar definiciones de seguridad

En matemáticas, una definición realmente no puede ser "incorrecta", pero puede ser "no tan útil como se esperaba" o puede "no capturar adecuadamente tu intuición" sobre el concepto.

Las definiciones de seguridad son definiciones matemáticas. En este capítulo introdujimos dos definiciones de seguridad: una en el estilo "real-vs-aleatorio" y otra en el estilo "intercambiabilidad". En esta sección tratamos las *definiciones de seguridad en sí mismas* como objetos dignos de estudio. ¿Son ambas definiciones de seguridad "lo mismo", en cierto sentido? ¿Capturan ambas todas nuestras intuiciones sobre la seguridad?

5.7.1. Una definición de seguridad implica otra

Una forma de comparar/contrastar dos definiciones de seguridad es demostrar que una implica a la otra. En otras palabras, si un esquema de cifrado Σ satisface la definición #1, entonces también satisface la definición #2.

Teorema 5.7.1. *Si un esquema de cifrado Σ tiene textos cifrados uniformes de un solo uso (Definición 5.4.1), entonces Σ también tiene intercambiabilidad de un solo uso (Definición 5.4.2). En otras palabras:*

$$\mathcal{L}_{otu}^{\Sigma} \equiv \mathcal{L}_{otu\text{-}\$}^{\Sigma} \implies \mathcal{L}_{otint\text{-}0}^{\Sigma} \equiv \mathcal{L}_{otint\text{-}1}^{\Sigma}$$

Si te sientes cómodo con lo que significa toda la terminología, entonces el significado de esta afirmación es bastante simple y poco sorprendente. "Si todos los textos planos m inducen una distribución *uniforme* de textos cifrados, entonces todos los m inducen la *misma* distribución de textos cifrados".

[4] Yo diría que la librería "no compila" debido a un error de alcance/referencia.

Esta afirmación bastante directa se puede probar formalmente, lo que nos da otro ejemplo de la técnica de prueba de seguridad:

Demostración. Estamos probando una declaración "si-entonces". Queremos mostrar que la parte "entonces" de la declaración es verdadera; es decir, $\mathcal{L}^{\Sigma}_{\text{otint-0}} \equiv \mathcal{L}^{\Sigma}_{\text{otint-1}}$. Se nos permite usar el hecho de que la parte "si" es verdadera; es decir, $\mathcal{L}^{\Sigma}_{\text{otu}} \equiv \mathcal{L}^{\Sigma}_{\text{otu-\$}}$.

La prueba utiliza la técnica de librarías intermedias. Comenzaremos con la librería $\mathcal{L}^{\Sigma}_{\text{otint-0}}$ y realizaremos una pequeña secuencia de cambios justificables en ella, hasta llegar finalmente a $\mathcal{L}^{\Sigma}_{\text{otint-1}}$. En el camino, podemos usar el hecho de que $\mathcal{L}^{\Sigma}_{\text{otu}} \equiv \mathcal{L}^{\Sigma}_{\text{otu-\$}}$. Esto sugiere una "estrategia" para la prueba: si de alguna manera podemos lograr que $\mathcal{L}^{\Sigma}_{\text{otu}}$ aparezca como un componente en una de las librerías intermedias, entonces podemos reemplazarlo con $\mathcal{L}^{\Sigma}_{\text{otu-\$}}$ (o viceversa), de una manera que, con suerte, avance hacia nuestro objetivo de transformar $\mathcal{L}^{\Sigma}_{\text{otint-0}}$ en $\mathcal{L}^{\Sigma}_{\text{otint-1}}$.

A continuación, enumeramos la secuencia de librerías intermedias y justificamos por qué cada una es intercambiable con la librería anterior:

$\mathcal{L}^{\Sigma}_{\text{otint-0}}$

EAVESDROP(m_0, m_1):
$k \leftarrow \Sigma.\text{Gen}$
$c := \Sigma.\text{Enc}(k, m_0)$
return c

El punto de partida es la librería $\mathcal{L}^{\Sigma}_{\text{otint-0}}$.

EAVESDROP(m_0, m_1):
$c :=$ CTXT(m_0)
return c

$\diamond$

$\mathcal{L}^{\Sigma}_{\text{otu}}$

CTXT(m):
$k \leftarrow \Sigma.\text{Gen}$
$c := \Sigma.\text{Enc}(k, m)$
return c

Factorizar un bloque de sentencias en una subrutina hace posible escribir la librería como una compuesta, pero no afecta su comportamiento externo. Observa que la nueva subrutina es exactamente la librería $\mathcal{L}^{\Sigma}_{\text{otu}}$ de la Definición 5.4.1. Esta fue una elección estratégica, debido a lo que sucede a continuación.

EAVESDROP(m_0, m_1):
$c :=$ CTXT(m_0)
return c

$\diamond$

$\mathcal{L}^{\Sigma}_{\text{otu-\$}}$

CTXT(m):
$c \leftarrow \Sigma.\mathcal{C}$
return c

$\mathcal{L}^{\Sigma}_{\text{otu}}$ ha sido reemplazado con $\mathcal{L}^{\Sigma}_{\text{otu-\$}}$. El Lema 5.6.1 dice que este cambio no tiene ningún efecto en el comportamiento de la librería, ya que las dos librerías $\mathcal{L}^{\Sigma}_{\text{otu}}$ y $\mathcal{L}^{\Sigma}_{\text{otu-\$}}$ son intercambiables.

EAVESDROP(m_0, m_1):
$c :=$ CTXT(m_1)
return c

$\diamond$

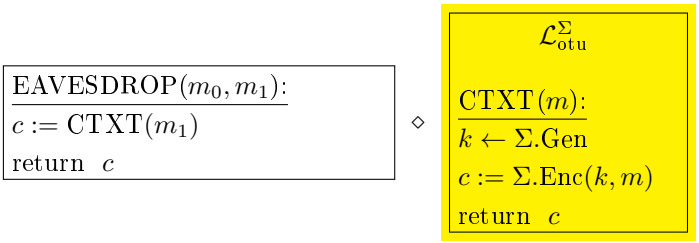
$\mathcal{L}^{\Sigma}_{\text{otu-\$}}$

CTXT(m):
$c \leftarrow \Sigma.\mathcal{C}$
return c

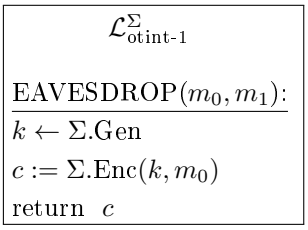
El argumento para CTXT se ha cambiado de m_0 a m_1. Esto no tiene ningún efecto en el comportamiento de la librería, ¡ya que CTXT en realidad no usa su argumento en estas librerías!

La transición anterior es la más importante de la prueba, ya que proporciona una idea de cómo se nos ocurrió esta secuencia particular de librerías intermedias. Al observar los puntos finales deseados de nuestra secuencia de librerías, $\mathcal{L}^{\Sigma}_{\text{otint-0}}$ y $\mathcal{L}^{\Sigma}_{\text{otint-1}}$, vemos que difieren solo en el intercambio de m_0 por m_1. Si no nos sentimos cómodos con simplemente "observar" las cosas, nos gustaría una mejor justificación de por qué es "seguro" intercambiar m_0 por m_1 (es decir, por qué no tiene ningún efecto en el comportamientod de cualquier programa que se vincula a ellas). Sin embargo, la propiedad de textos cifrados uniformes muestra que $\mathcal{L}^{\Sigma}_{\text{otint-0}}$ de hecho tiene el mismo comportamiento que una librería $\mathcal{L}^{\Sigma}_{\text{hyb}}$ que no usa ni m_0 ni m_1. ¡En un programa que no usa m_0 ni m_1, está claro que podemos intercambiarlos!

Habiendo realizado este cambio crucial, ahora podemos ejecutar la misma secuencia de pasos, pero en sentido inverso.



$\mathcal{L}^{\Sigma}_{\text{otu-\$}}$ ha sido reemplazado con $\mathcal{L}^{\Sigma}_{\text{otu}}$. Esta es otra aplicación del Lema 5.6.1.



Se ha incluido el código de una subrutina en el lugar donde se ha llamado (*inlining*), lo cual no tiene ningún efecto en el comportamiento de la librería. El resultado es exactamente $\mathcal{L}^{\Sigma}_{\text{otint-1}}$.

Juntando todo, mostramos que $\mathcal{L}^{\Sigma}_{\text{otint-0}} \equiv \mathcal{L}^{\Sigma}_{\text{hyb-1}} \equiv \cdots \equiv \mathcal{L}^{\Sigma}_{\text{hyb-4}} \equiv \mathcal{L}^{\Sigma}_{\text{otint-1}}$. Esto completa la prueba, y concluimos que Σ satisface la definición de intercambiabilidad de un solo uso.

■

5.7.2. Una definición de seguridad no implica otra

Otra forma en que podríamos comparar las definiciones de seguridad es identificar cualquier esquema que satisfaga una definición sin satisfacer la otra. Esto nos ayuda a comprender los límites y los "casos límite" de la definición.

Una advertencia: si tenemos dos definiciones de seguridad que capturan bastante bien nuestras intuiciones, entonces cualquier esquema que satisfaga una definición y no la otra seguramente parecerá *antinatural* y *artificioso*. El punto es obtener una mayor comprensión de las definiciones de seguridad en sí mismas, y los esquemas antinaturales/artificiosos son solo un medio para hacerlo.

Teorema 5.7.2. *Existe un esquema de cifrado que intercambiabilidad de un solo uso (Definición 5.4.2) pero no tiene textos cifrados uniformes de un solo uso (Definición 5.4.1). En otras palabras, intercambiabilidad de un solo uso no implica necesariamente textos cifrados uniformes de un solo uso.*

Demostración. A continuación, se presenta un esquema de cifrado de este tipo:

	Gen:	Enc(k, m):	Dec(k, c):
$\mathcal{K} = \{0,1\}^\lambda$	$k \leftarrow \{0,1\}^\lambda$	$c' := k \oplus m$	$c' :=$ primeros λ bits de c
$\mathcal{M} = \{0,1\}^\lambda$	return k	$c := c' \| \| 00$	$m := k \oplus c'$
$\mathcal{C} = \{0,1\}^{\lambda+2}$		return c	return m

Este esquema es simplemente OTP con los bits 00 añadidos a cada texto cifrado. Los siguientes hechos sobre el esquema deberían ser creíbles (y deberías intentar demostrarlos formalmente si deseas más práctica en ese tipo de cosas):

- Este esquema tiene intercambiabilidad de un solo uso, lo que significa que los cifrados de m_0 se distribuyen de forma idéntica a los cifrados de m_1, para cualquier m_0 y m_1 a elección del atacante. Podemos caracterizar la distribución del texto cifrado en ambos casos como "λ bits uniformes seguidos de 00". ¡Piensa en cómo podrías usar la técnica de demostración con librerías para probar formalmente que este esquema tiene intercambiabilidad de un solo uso!
- Este esquema no satisface la propiedad de textos cifrados uniformes. Sus textos cifrados siempre terminan en 00, mientras que las cadenas uniformes terminan en 00 con una probabilidad de 1/4. ¡Piensa en cómo podrías formalizar esta observación construyendo un distinguidor para las dos librerías relevantes!

■

Podrías estar pensando, seguramente esto se puede solucionar redefiniendo el espacio de textos cifrados $\mathcal{C}$ como el conjunto de cadenas de $\lambda + 2$ bits cuyos dos últimos bits son 00. Esta es una idea inteligente, y de hecho funcionaría. Si cambiamos la definición del espacio de textos cifrados $\mathcal{C}$ siguiendo esta sugerencia, entonces el esquema satisfaría la propiedad de textos cifrados uniformes (esto se debe a que la librería $\mathcal{L}^{\Sigma}_{\text{ot u-\$}}$ muestrea uniformemente de lo que sea que se especifique como $\mathcal{C}$ como parte del esquema de cifrado).

Pero esta observación plantea un punto interesante. ¿No es extraño que la seguridad dependa de lo estrictamente que se defina el conjunto $\mathcal{C}$ de textos cifrados, cuando $\mathcal{C}$ realmente no tiene ningún efecto en la funcionalidad del cifrado? De nuevo, a nadie le importa realmente este artificioso esquema de cifrado "OTP + 00". ¡El punto es iluminar casos límite interesantes en la propia definición de seguridad!

Capítulo 6

Basando la criptografía en cálculos intratables

John Nash fue un matemático que ganó el Premio Nobel de Economía en 1994 por su trabajo en teoría de juegos. La historia de su vida se convirtió en una exitosa película, *Una mente maravillosa*. En 1955, Nash mantuvo correspondencia con la Agencia de Seguridad Nacional de los Estados Unidos (NSA)[1], discutiendo nuevos métodos de cifrado que había ideado. En estas cartas, también propone algunos principios generales de la criptografía (el resaltado en negrita no está en el original):

> ... en principio, el enemigo necesita muy poca información para comenzar a desglosar el proceso. Esencialmente, tan pronto como se han transmitido λ bits[2] del mensaje cifrado, la clave está casi determinada. Esto no es seguridad, ya que una clave práctica no debería ser demasiado larga. **Pero esto no considera lo fácil o difícil que es para el enemigo realizar el cálculo para determinar la clave. Si este cálculo, aunque posible en principio, fuera suficientemente largo en el mejor de los casos, entonces el proceso aún podría ser seguro en un sentido práctico.**

Nash está diciendo algo bastante profundo: realmente no importa si los ataques son *imposibles*, solo si los ataques son *computacionalmente inviables*. Si sus cartas no hubieran permanecido clasificadas hasta 2012, podrían haber acelerado el desarrollo de la criptografía "moderna", en la que la seguridad se basa en cálculos intratables. Tal como están las cosas, se adelantó décadas a su tiempo al identificar uno de los conceptos más importantes de la criptografía moderna.

[1] Las cartas originales, escritas a mano por Nash, están disponibles en: https://www.nsa.gov/Portals/70/documents/news-features/declassified-documents/nash-letters/nash_letters1.pdf.

[2] Nash originalmente utilizó r para denotar la longitud de la clave, en bits. En todos los extractos citados en este capítulo, he traducido sus expresiones matemáticas a nuestra notación (λ).

6.1. ¿Qué califica como un ataque "computacionalmente inviable"?

Esquemas como el one-time pad no se pueden romper, ni siquiera por un atacante que realice un ataque de fuerza bruta, probando todas las claves posibles. Sin embargo, todos los esquemas futuros que veremos sí se pueden romper por tal ataque. Nash señala que, para un esquema con claves de λ bits:

> El procedimiento de cálculo más directo sería que el enemigo probara las 2^λ claves posibles, una por una. Obviamente, esto fácilmente se vuelve inviable para el enemigo simplemente eligiendo un λ lo suficientemente grande.

Llamamos a λ el **parámetro de seguridad** del esquema. Es como una perilla que permite al usuario ajustar la seguridad a cualquier nivel deseado. Aumentar λ hace que la dificultad de un ataque de fuerza bruta crezca exponencialmente rápido. Idealmente, cuando se usan claves de λ bits, cada ataque (no solo un ataque de fuerza bruta) tendrá una dificultad de aproximadamente 2^λ. Sin embargo, a veces los ataques más rápidos son inevitables. Más adelante en este capítulo, veremos por qué muchos esquemas con claves de λ bits tienen ataques que cuestan solo $2^{\lambda/2}$. Es común ver un esquema descrito como que tiene **seguridad de n bits** si el mejor ataque conocido requiere 2^n pasos.

¿Cuán inviable es un cálculo de fuerza bruta en una clave de 64 bits? ¿Una clave de 128 bits? Los números enormes como 2^{64} y 2^{128} son difíciles de comprender a un nivel intuitivo.

Ejemplo 6.1. Puede ser útil pensar en el costo de un cálculo en términos de valor monetario, y una forma conveniente de asignar tales costos monetarios es usar el modelo de precios de un proveedor de computación en la nube. A continuación, he calculado aproximadamente cuánto costaría un cálculo que involucra 2^λ ciclos de CPU en Amazon EC2, para varias opciones de λ.

ciclos de reloj	costo aprox.	referencia
2^{50}	\$3.50	un bocata de jamón ibérico
2^{55}	\$100	dos entradas decentes para El Rey León
2^{65}	\$130,000	precio medio de un piso de 90m^2 (4° sin ascensor) en Móstoles
2^{75}	\$130 millones	presupuesto de una de las películas de Harry Potter
2^{85}	\$140 mil millones	PIB de Hungría
2^{92}	\$2 billones	PIB de los Estados Unidos
2^{99}	\$2 cuatrillones	toda la actividad económica humana desde el 300,000 a. C.
2^{128}	realmente mucho	el esfuerzo de mil millones de civilizaciones humanas

Recuerda, esta tabla solo muestra el costo para realizar 2^λ ciclos de reloj. Un ataque de fuerza bruta que comprueba 2^λ claves requeriría muchos más ciclos que eso. Pero, como descargo de responsabilidad, estos números reflejan solo el costo minorista de realizar un cálculo, en hardware de propósito general bastante estándar. Una organización gubernamental sería capaz de fabricar hardware de propósito especial que reduciría significativamente el costo del cálculo.

6.1.1. Tiempo asintótico de ejecución

Es instructivo pensar en el costo monetario de un cálculo enorme, pero no necesariamente nos ayuda a trazar la línea entre ataques "factibles" (de los que queremos protegernos) e "infactibles" (de los que acordamos que no necesitamos preocuparnos). Necesitamos poder trazar tal línea para hacer definiciones de seguridad que digan "solo se excluyen los ataques factibles".

Una vez más, John Nash pensó en esta cuestión. Sugirió considerar el costo *asintótico* de un ataque: ¿cómo escala el costo de un cálculo a medida que el parámetro de seguridad λ tiende a infinito?

> Por lo tanto, una forma lógica de clasificar los procesos de cifrado es por la forma en que la longitud del cálculo para la computación de la clave aumenta con el aumento de la longitud de la clave. Esto es, en el mejor de los casos, exponencial y, en el peor de los casos, probablemente una pequeña potencia de λ, λ^2 o λ^3, como en los cifrados de sustitución.

Nash destaca la importancia de los ataques que se ejecutan en tiempo polinómico:

Definición 6.1.1. *Un programa se ejecuta en tiempo polinómico si existe una constante $c > 0$ tal que para todas las cadenas de entrada x suficientemente largas, el programa se detiene después de no más de $O(|x|^c)$ pasos.*

Los algoritmos de tiempo polinómico escalan razonablemente bien (especialmente cuando el exponente es pequeño), pero los algoritmos de tiempo exponencial no. Probablemente no sea una sorpresa para los lectores modernos ver "tiempo polinómico" como sinónimo de "eficiente". Sin embargo, vale la pena señalar que, una vez más, Nash se adelanta a su tiempo en relación con el campo de la informática.

En el contexto de la criptografía, nuestro objetivo será asegurar que ningún ataque de tiempo polinómico pueda romper la seguridad con éxito. No nos preocuparemos por ataques como la fuerza bruta que requieren tiempo exponencial.

El tiempo polinómico no es una coincidencia perfecta con lo que queremos decir cuando hablamos informalmente de algoritmos "eficientes". Los algoritmos con tiempo de ejecución $\Theta(n^{1000})$ son técnicamente de tiempo polinómico, mientras que aquellos con tiempo de ejecución $O(n^{\log\log\log n})$ no lo son. A pesar de ello, el tiempo polinómico es extremadamente útil debido a la siguiente *propiedad de clausura*: repetir un proceso de tiempo polinómico un número polinómico de veces resulta en un proceso de tiempo polinómico en general.

6.1.2. Trampa potencial: algoritmos numéricos

Cuando estudiemos la criptografía de clave pública, discutiremos algoritmos que operan con números muy grandes (por ejemplo, de miles de bits de longitud). Debes recordar que representar el número N en una computadora requiere solo $\sim \log_2 N$ bits. ¡Esto significa que $\log_2 N$, en lugar de N, es nuestro parámetro de seguridad! Por lo tanto, estaremos interesados en si ciertas operaciones sobre el número N se ejecutan en tiempo polinómico como una función de $\log_2 N$, en lugar de en N. Ten en cuenta que la diferencia entre un tiempo de ejecución $O(\log N)$ y $O(N)$ es la diferencia entre escribir un número y contar hasta el número.

Como referencia, aquí hay algunas operaciones numéricas que utilizaremos más adelante en la clase y sus eficiencias conocidas:

Algoritmo eficiente conocido:	**Ningún algoritmo eficiente conocido:**
Cálculo de MCDs	Factorización de enteros
Aritmética módulo N	Cálculo de $\phi(N)$ dado N
Inversas módulo N	Logaritmo discreto
Exponenciación módulo N	Raíces cuadradas módulo N compuesto

Nuevamente, "eficiente" significa tiempo polinómico. Además, solo consideramos algoritmos de tiempo polinómico que se ejecutan en computadoras *clásicas* estándar. De hecho, todos los problemas en la columna de la derecha *sí* tienen algoritmos conocidos de tiempo polinómico en computadoras *cuánticas*.

6.2. ¿Qué califica como una probabilidad de éxito "despreciable"?

No es suficiente considerar solo el tiempo de ejecución de un ataque. Por ejemplo, considere un atacante que simplemente intenta adivinar la clave secreta de una víctima, haciendo un solo intento. ¡Este ataque es extremadamente barato, pero aún tiene una probabilidad no nula de romper la seguridad!

Además del tiempo de ejecución de un ataque, también necesitamos considerar su probabilidad de éxito. No queremos preocuparnos por ataques que sean tan costosos como un ataque de fuerza bruta, y no queremos preocuparnos por ataques cuya probabilidad de éxito sea tan baja como un ataque de adivinanza ciega.

Un ataque con una probabilidad de éxito de 2^{-128} realmente no debería contar como un ataque, pero un ataque con una probabilidad de éxito de $1/2$ sí debería. En algún punto entre 2^{-128} y 2^{-1} necesitamos encontrar un lugar razonable para trazar una línea.

Ejemplo 6.2. Ahora estamos tratando con probabilidades extremadamente pequeñas que pueden ser difíciles de visualizar. Nuevamente, puede ser útil conceptualizar estas probabilidades con una referencia más familiar:

probabilidad	**equivalente**
2^{-10}	full house en poker de 5 cartas
2^{-20}	royal flush en poker de 5 cartas
2^{-28}	ganas el gordo en la lotería de Navidad
2^{-40}	royal flush en 2 partidas consecutivas de poker
2^{-60}	el próximo meteorito que golpee la Tierra aterriza en este cuadrado → ■

Como antes, no está claro exactamente dónde trazar la línea entre una probabilidad de éxito "razonable" y "no razonable" para un ataque. Al igual que hicimos con el tiempo de ejecución polinómico, también podemos usar un enfoque asintótico para definir cuándo una probabilidad es despreciablemente pequeña. Así como el "tiempo polinómico" considera qué tan rápido el tiempo de ejecución de un algoritmo se acerca al infinito a medida que crece su entrada, también podemos considerar qué tan rápido una probabilidad de éxito se acerca a cero a medida que crece el parámetro de seguridad.

En un esquema con claves de λ bits, un ataque de adivinanza ciega tiene éxito con una probabilidad de $1/2^{\lambda}$. Ahora, ¿qué pasa con un atacante que hace 2 adivinanzas ciegas, o λ adivinanzas, o λ^{42} adivinanzas? Tal atacante aún se ejecutaría en tiempo polinómico y tiene una probabilidad de éxito de $2/2^{\lambda}$, $\lambda/2^{\lambda}$ o $\lambda^{42}/2^{\lambda}$. Sin embargo, sin importar qué polinomio pongas en el numerador, la probabilidad aún tiende a cero.

De hecho, $1/2^\lambda$ se acerca a cero tan rápido que ningún polinomio puede "rescatarlo"; o, en otras palabras, se acerca a cero más rápido que 1 sobre cualquier polinomio. Esta idea nos lleva a nuestra definición formal:

Definición 6.2.1. *Una función f es **despreciable** si, para cada polinomio p, tenemos $\lim_{\lambda\to\infty} p(\lambda)f(\lambda) = 0$.*

En otras palabras, una función despreciable se acerca a cero tan rápido que nunca se le puede alcanzar multiplicándola por un polinomio. Esta es exactamente la propiedad que queremos de una garantía de seguridad que se supone que se cumple contra todos los adversarios de tiempo polinómico. Si un atacante de tiempo polinómico tiene éxito con una probabilidad f, entonces repetir el mismo ataque p veces independientes seguiría siendo un ataque general de tiempo polinómico (si p es un polinomio), y su probabilidad de éxito sería $p \cdot f$.

Cuando quieres verificar si una función es despreciable, solo tienes que considerar polinomios p de la forma $p(\lambda) = \lambda^c$ para alguna constante c:

Afirmación 6.2.2. *Si para cada entero c, $\lim_{\lambda\to\infty} \lambda^c f(\lambda) = 0$, entonces f es despreciable.*

Demostración. Supongamos que f tiene esta propiedad y tomemos un polinomio arbitrario p. Queremos demostrar que $\lim_{\lambda\to\infty} p(\lambda)f(\lambda) = 0$.

Si d es el grado de p, entonces $\lim_{\lambda\to\infty} \frac{p(\lambda)}{\lambda^{d+1}} = 0$. Por lo tanto,

$$\lim_{\lambda\to\infty} p(\lambda)f(\lambda) = \lim_{\lambda\to\infty} \left[\frac{p(\lambda)}{\lambda^{d+1}}(\lambda^{d+1}\cdot f(\lambda))\right] = \left(\lim_{\lambda\to\infty} \frac{p(\lambda)}{\lambda^{d+1}}\right)\left(\lim_{\lambda\to\infty} \lambda^{d+1}\cdot f(\lambda)\right) = 0\cdot 0.$$

La segunda igualdad es una propiedad válida de los límites, ya que los dos límites de la derecha existen y no son una expresión indeterminada como $0 \cdot \infty$. La igualdad final se deduce de la hipótesis sobre f. ■

Ejemplo 6.3. La función $f(\lambda) = 1/2^\lambda$ es despreciable, ya que para cualquier entero c, tenemos:

$$\lim_{\lambda\to\infty} \lambda^c/2^\lambda = \lim_{\lambda\to\infty} 2^{c\log(\lambda)}/2^\lambda = \lim_{\lambda\to\infty} 2^{c\log(\lambda)-\lambda} = 0$$

ya que $c\log(\lambda) - \lambda$ tiende a $-\infty$ en el límite, para cualquier constante c. Usando un razonamiento similar, se puede demostrar que las siguientes funciones también son despreciables:

$$\frac{1}{2^{\lambda/2}}, \quad \frac{1}{2^{\sqrt{\lambda}}}, \quad \frac{1}{2^{\log^2\lambda}}, \quad \frac{1}{\lambda^{\log\lambda}}.$$

Funciones como $1/\lambda^5$ se acercan a cero, pero no lo suficientemente rápido como para ser despreciables. Para ver por qué, podemos tomar el polinomio $p(\lambda) = \lambda^6$ y ver que el límite resultante no satisface el requisito de la Definición 6.2.1:

$$\lim_{\lambda\to\infty} p(\lambda)\frac{1}{\lambda^5} = \lim_{\lambda\to\infty} \lambda = \infty \neq 0$$

En esta asignatura, cuando veamos una función despreciable, normalmente siempre será una que sea fácil de reconocer como despreciable (al igual que realmente no encontrarás algoritmos donde sea difícil determinar si el tiempo de ejecución es polinómico).

Definición 6.2.3. *Si $f, g : \mathbb{N} \to \mathbb{R}$ son dos funciones, escribimos $f \approx g$ para indicar que $|f(\lambda) - g(\lambda)|$ es una función despreciable.*

Usamos la terminología de funciones despreciables exclusivamente cuando discutimos probabilidades, por lo que lo siguiente es común:

$\Pr[X] \approx 0$	$\Leftrightarrow$	el evento X casi nunca ocurre
$\Pr[Y] \approx 1$	$\Leftrightarrow$	el evento Y casi siempre ocurre
$\Pr[A] \approx \Pr[B]$	$\Leftrightarrow$	los eventos A y B ocurren esencialmente con la misma probabilidad [3]

Además, el símbolo $\approx$ es *transitivo*[4]: si $\Pr[X] \approx \Pr[Y]$ y $\Pr[Y] \approx \Pr[Z]$, entonces $\Pr[X] \approx \Pr[Z]$ (quizás con una diferencia ligeramente mayor, pero aún despreciable).

6.3. Indistinguibilidad

Hasta ahora, hemos estado escribiendo definiciones de seguridad formales en términos de bibliotecas intercambiables, lo que requiere que dos bibliotecas tengan exactamente el mismo efecto en cada programa que las llama. En adelante, nuestras definiciones de seguridad no serán tan exigentes. Primero, solo consideramos programas de llamada de tiempo polinómico; segundo, no requerimos que las bibliotecas tengan exactamente el mismo efecto en el programa que las llama, solo que la diferencia en los efectos sea despreciable.

Definición 6.3.1. *Sean $\mathcal{L}_0$ y $\mathcal{L}_1$ dos librerías con una interfaz común. Decimos que $\mathcal{L}_0$ y $\mathcal{L}_1$ son **indistinguibles**, y escribimos $\mathcal{L}_0 \approx \mathcal{L}_1$, si para todos los programas de tiempo polinómico $\mathcal{A}$ que producen un solo bit, $\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1] \approx \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1]$.*

*Llamamos a la cantidad $|\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1]|$ la **ventaja** o **sesgo** de $\mathcal{A}$ al distinguir $\mathcal{L}_0$ de $\mathcal{L}_1$. Por lo tanto, dos librerías son indistinguibles si todos los programas $\mathcal{A}$ de tiempo polinómico tienen una ventaja despreciable al distinguirlas.*

De las propiedades del símbolo "$\approx$", podemos ver que la indistinguibilidad de las librerías también es transitiva, lo que nos permite llevar a cabo pruebas de seguridad de la misma manera que antes.

Ejemplo 6.4. Aquí hay un ejemplo muy simple de dos librerías indistinguibles:

$\mathcal{L}_0$

PREDICT(x):
$s \leftarrow \{0,1\}^\lambda$
return $x \overset{?}{=} s$

$\mathcal{L}_1$

PREDICT(x):
return false

Imagina un programa intentando predecir qué cadena se elegirá al muestrear uniformemente de $\{0,1\}^\lambda$. La librería izquierda le dice al programa si su predicción fue correcta. La librería derecha ni siquiera se molesta en muestrear una cadena, simplemente siempre dice "lo siento, tu predicción fue incorrecta".

[3] $\Pr[A] \approx \Pr[B]$ no significa que los eventos A y B ocurran casi siempre juntos (cuando A y B están definidos sobre un mismo espacio de probabilidad). Por ejemplo, imagina que A es el evento "la moneda cayó cara" B es el evento "la moneda cayó cruz". Estos eventos tienen la misma probabilidad, pero nunca ocurren juntos. Para afirmar que "A y B ocurren casi siempre juntos", tendrías que decir algo como $\Pr[A \oplus B] = 0$, donde $A \oplus B$ denota el evento en el que ocurre exactamente uno de A o B.

[4] Es transitiva solo cuando se aplica un número polinómico de veces. Por lo tanto, no puedes definir una serie completa de eventos X_i, demostrar que $\Pr[X_i] \approx \Pr[X_{i+1}]$ y concluir que $\Pr[X_1] \approx \Pr[X_{2^n}]$. Es raro que encontremos esta sutileza en esta asignatura.

Aquí hay una estrategia obvia (quizás no la mejor, ya veremos) para distinguir estas librerías. El programa $\mathcal{A}_{\text{obvio}}$ llama a PREDICT muchas veces y produce 1 si alguna vez recibió verdadero como respuesta. Dado que parece que el argumento para PREDICT podría no tener ningún efecto, usemos la cadena de todos ceros como argumento cada vez.

$\mathcal{A}_{\text{obvio}}$

```
do q veces:
    if PREDICT(0^λ) = true
      return 1
    end if
end do
return 0
```

- $\mathcal{L}_1$ nunca puede dar verdadero, por lo que $\Pr[\mathcal{A}_{\text{obvio}} \diamond \mathcal{L}_1 \Rightarrow 1] = 0$.
- En $\mathcal{L}_0$ cada llamada a PREDICT tiene una probabilidad independiente $1/2^\lambda$ de dar verdadero. Así que $\Pr[\mathcal{A}_{\text{obvio}} \diamond \mathcal{L}_0 \Rightarrow 1]$ seguramente es distinto de cero. En realidad, la probabilidad exacta es un poco engorrosa de escribir:

$$\begin{aligned}\Pr[\mathcal{A}_{\text{obvio}} \diamond \mathcal{L}_0 \Rightarrow 1] &= 1 - \Pr[\mathcal{A}_{\text{obvio}} \diamond \mathcal{L}_0 \Rightarrow 0] \\ &= 1 - \Pr[\text{todas las } q \text{ llamadas independientes a PREDICT dan falso}] \\ &= 1 - \left(1 - \frac{1}{2^\lambda}\right)^q\end{aligned}$$

  En lugar de entender esta probabilidad, podemos simplemente calcular un límite superior para ella. Usando la unión de probabilidad, obtenemos:

$$\begin{aligned}\Pr[\mathcal{A}_{\text{obvio}} \diamond \mathcal{L}_0 \Rightarrow 1] \leq{}& \Pr[\text{la primera llamada a PREDICT da verdadero}] \\ &+ \Pr[\text{la segunda llamada a PREDICT retorna verdadero}] + \ldots \\ ={}& q\frac{1}{2^\lambda}.\end{aligned}$$

  Esta es una sobreestimación de algunas probabilidades (por ejemplo, si la primera llamada a PREDICT da verdadero, entonces la segunda llamada no se realiza). Más fundamentalmente, $q/2^\lambda$ excede 1 cuando q es grande. Pero, sin embargo, $\Pr[\mathcal{A}_{\text{obvio}} \diamond \mathcal{L}_0 \Rightarrow 1] < q/2^\lambda$.

Hemos demostrado que $\mathcal{A}_{\text{obvio}}$ tiene una ventaja no nula. Esto es suficiente para demostrar que $\mathcal{L}_0 \not\equiv \mathcal{L}_1$. También hemos demostrado que $\mathcal{A}_{\text{obvio}}$ tiene una ventaja de como máximo $q/2^\lambda$. Dado que $\mathcal{A}_{\text{obvio}}$ se ejecuta en tiempo polinómico, solo puede realizar un número polinómico q de consultas a la librería, por lo que $q/2^\lambda$ es despreciable. Sin embargo, esto no es suficiente para demostrar que $\mathcal{L}_0 \approx \mathcal{L}_1$, ya que solo considera un único programa. Para demostrar que las librerías son indistinguibles, debemos demostrar que la ventaja de **todo** programa que llama es despreciable.

En unas pocas páginas, demostraremos que para *cualquier* $\mathcal{A}$ que realiza q llamadas a PREDICT,

$$|\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1]| \leq \frac{q}{2^\lambda}.$$

Para cualquier $\mathcal{A}$ polinómico, el número q de llamadas a PREDICT será un polinomio en λ, lo que hace que $q/2^\lambda$ sea una función despreciable. Por lo tanto, $\mathcal{L}_0 \approx \mathcal{L}_1$.

6.3.1. Otras propiedades

Lema 6.3.2. *Si $\mathcal{L}_1 \approx \mathcal{L}_2$ entonces $\mathcal{L}_2 \approx \mathcal{L}_1$. También, si $\mathcal{L}_1 \approx \mathcal{L}_2 \approx \mathcal{L}_3$ entonces $\mathcal{L}_1 \approx \mathcal{L}_3$.*

Análogo al Lema 5.6.1, también tenemos el siguiente lema de encadenamiento de librerías:

Lema 6.3.3. *Si $\mathcal{L}_0 \approx \mathcal{L}_1$ entonces $\mathcal{L}^* \diamond \mathcal{L}_0 \approx \mathcal{L}^* \diamond \mathcal{L}_1$ para cualquier librería $\mathcal{L}^*$ de tiempo polinómico.*

6.3.2. Lema del suceso malo

Una situación común es cuando se espera que dos librerías ejecuten exactamente las mismas sentencias, hasta que ocurre alguna condición rara y excepcional. En ese caso, podemos acotar la ventaja de distinción de un atacante mediante la probabilidad de la condición excepcional.

Más formalmente,

Lema 6.3.4. *Sean $\mathcal{L}_0$ y $\mathcal{L}_1$ librerías que definen cada una una variable llamada 'bad' que se inicializa a 0. Si $\mathcal{L}_0$ y $\mathcal{L}_1$ tienen código idéntico, excepto por bloques de código alcanzables solo cuando bad = 1 (p. ej., protegidos por una sentencia ïf bad = 1"), entonces para todos los programas de tiempo polinómico $\mathcal{A}$ que producen un solo bit,*

$$|\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1]| \leq \Pr[\mathcal{A} \diamond \mathcal{L}_0 \text{ establece } bad = 1].$$

Demostración. Fijemos un programa de llamada arbitrario $\mathcal{A}$. En esta demostración, usamos probabilidades condicionales[5] para aislar los casos donde bad se cambia a 1. Definimos los siguientes eventos:

- B_0: el evento de que $\mathcal{A} \diamond \mathcal{L}_0$ establece bad a 1 en algún momento.
- B_1: el evento de que $\mathcal{A} \diamond \mathcal{L}_1$ establece bad a 1 en algún momento.

También escribimos $\overline{B_0}$ y $\overline{B_1}$ para denotar los eventos complementarios correspondientes. A partir de la probabilidad condicional, podemos escribir:

$$\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1] = \Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1 \mid B_0] \cdot \Pr[B_0] + \Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1 \mid \overline{B_0}] \cdot \Pr[\overline{B_0}]$$

$$\Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1] = \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1 \mid B_1] \cdot \Pr[B_1] + \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1 \mid \overline{B_1}] \cdot \Pr[\overline{B_1}]$$

Nuestra primera observación es que $\Pr[B_0] = \Pr[B_1]$. Esto se debe a que en el momento en que bad se cambia a 1 por primera vez, la librería solo ha estado ejecutando instrucciones que son las mismas en $\mathcal{L}_0$ y $\mathcal{L}_1$. En otras palabras, la decisión de establecer bad a 1 está determinada por la misma secuencia de instrucciones en ambas librerías, por lo que ocurre con la misma probabilidad en ambas librería.

[5] El uso de probabilidades condicionales aquí es delicado y propenso a errores sutiles. Para una discusión sobre los errores, consulte el artículo donde apareció este lema por primera vez: Mihir Bellare & Phillip Rogaway: Çode-Based Game-Playing Proofs and the Security of Triple Encryption,.ën Eurocrypt 2006. ia.cr/2004/331

Como notación abreviada, definimos $p^* \stackrel{\text{def}}{=} \Pr[B_0] = \Pr[B_1]$. Entonces podemos escribir la ventaja de $\mathcal{A}$ como:

$$\begin{aligned}\mathtt{Adv}(\mathcal{A}) &= |\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1]| \\ &= \left| \begin{array}{r} \big(\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1 \mid B_0] \cdot p^* + \Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1 \mid \overline{B_0}](1-p^*)\big) \\ -\big(\Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1 \mid B_1] \cdot p^* + \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1 \mid \overline{B_1}](1-p^*)\big) \end{array} \right| \\ &= \left| \begin{array}{r} p^* \left(\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1 \mid B_0] - \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1 \mid B_1]\right) \\ +(1-p^*)\left(\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1 \mid \overline{B_0}] - \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1 \mid \overline{B_1}]\right) \end{array} \right|\end{aligned}$$

En ambas expresiones $\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1 \mid \overline{B_0}]$ y $\Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1 \mid \overline{B_1}]$, estamos condicionando a que bad nunca se establezca en 1. En este caso, ambas librerías están ejecutando la misma secuencia de instrucciones, por lo que las probabilidades son iguales (y la diferencia de las probabilidades es cero). Sustituyendo, obtenemos:

$$\mathtt{Adv}(\mathcal{A}) = p^* \, |\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow 1 \mid B_0] - \Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow 1 \mid B_1]|$$

Intuitivamente, la demostración está confirmando la idea de que las diferencias solo se pueden notar entre $\mathcal{L}_0$ y $\mathcal{L}_1$ cuando bad se establece en 1 (correspondiente a nuestra condición sobre B_0 y B_1).

La cantidad dentro del valor absoluto es la diferencia de dos probabilidades, por lo que lo mayor que puede ser es 1. Por lo tanto,

$$\mathtt{Adv}(\mathcal{A}) \leq p^* \stackrel{\text{def}}{=} \Pr[B_0] = \Pr[\mathcal{A} \diamond \mathcal{L}_0 \text{ establece bad} = 1].$$

Esto completa la demostración. ■

Ejemplo 6.5. Consideremos $\mathcal{L}_0$ y $\mathcal{L}_1$ del ejemplo anterior (donde el programa $\mathcal{A}_{\text{obvio}}$ intenta "predecir" el resultado de muestrear uniformemente una cadena de λ bits). Podemos probar que son indistinguibles con la siguiente secuencia de librerías:

$\mathcal{L}_0$
PREDICT(x):
$s \leftarrow \{0,1\}^\lambda$
return $x \stackrel{?}{=} s$

$\equiv$

$\mathcal{L}_{\text{hyb-0}}$
bad:=0
PREDICT(x):
$s \leftarrow \{0,1\}^\lambda$
if $x = s$:
  bad:=1
  return true
return false

$\approx$

$\mathcal{L}_{\text{hyb-1}}$
bad:=0
PREDICT(x):
$s \leftarrow \{0,1\}^\lambda$
if $x = s$:
  bad:=1
return false

$\equiv$

$\mathcal{L}_1$
PREDICT(x):
return false

Justifiquemos cada uno de los pasos:

- $\mathcal{L}_0 \equiv \mathcal{L}_{\text{hyb-0}}$: La única diferencia es que $\mathcal{L}_{\text{hyb-0}}$ contiene una variable bad. Dado que en realidad nunca lee de esta variable, el cambio no puede tener ningún efecto.

- $\mathcal{L}_{\text{hyb-0}}$ y $\mathcal{L}_{\text{hyb-1}}$ difieren solo en la línea resaltada, a la que solo se puede llegar cuando bad $= 1$. Por lo tanto, del Lema 6.3.4:

$$|\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-0}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-1}} \Rightarrow 1]| \leq \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-0}} \text{ establece bad} = 1].$$

  Pero $\mathcal{A} \diamond \mathcal{L}_{\text{hyb-0}}$ solo establece bad $= 1$ si $\mathcal{A}$ predice exitosamente s en una de las llamadas a PREDICT. Con q llamadas a PREDICT, la probabilidad total de que esto suceda es a lo sumo $q/2^{\lambda}$, lo cual es despreciable cuando $\mathcal{A}$ se ejecuta en tiempo polinómico. Por lo tanto, $\mathcal{L}_{\text{hyb-0}} \approx \mathcal{L}_{\text{hyb-1}}$.

- $\mathcal{L}_{\text{hyb-1}} \equiv \mathcal{L}_1$: Similar a lo anterior, observe cómo las primeras 3 líneas de PREDICT en $\mathcal{L}_{\text{hyb-1}}$ en realidad no hacen nada. La subrutina va a devolver falso pase lo que pase. Ambas librerías tienen un comportamiento idéntico.

Dado que $\mathcal{L}_0 \equiv \mathcal{L}_{\text{hyb-0}} \approx \mathcal{L}_{\text{hyb-1}} \equiv \mathcal{L}_1$, esto prueba que $\mathcal{L}_0 \approx \mathcal{L}_1$.

6.4. Probabilidades de cumpleaños y muestreo con/sin reemplazo

En muchos esquemas criptográficos, los usuarios eligen repetidamente cadenas aleatorias (p. ej., cada vez que encriptan un mensaje), y la seguridad se rompe si la misma cadena se elige dos veces. Por lo tanto, es importante que la probabilidad de una muestra repetida sea *despreciable*. En esta sección calculamos la probabilidad de tales eventos y expresamos nuestros hallazgos de una manera modular, como una declaración sobre la indistinguibilidad de dos librerías.

6.4.1. Probabilidades de cumpleaños

Si q personas están en una habitación, ¿cuál es la probabilidad de que dos de ellas tengan el mismo cumpleaños (si asumimos que el cumpleaños de cada persona se elige uniformemente entre los posibles días de un año)? Esta pregunta se conoce como el *problema del cumpleaños*, y es famosa porque la respuesta es muy poco intuitiva para la mayoría de las personas.[6]

Hagamos la pregunta más general. Imaginemos tomar q muestras independientes y uniformes de un conjunto de N elementos. ¿Cuál es la probabilidad de que el mismo valor se elija más de una vez? En otras palabras, ¿cuál es la probabilidad de que el siguiente programa produzca 1?

$\mathcal{B}(q, N)$

```
for i := 1 to q:
    s_i ← {1, ..., N}
    for j := 1 to i − 1:
        if s_i = s_j then return 1
return 0
```

[6] A veces se le llama la "paradoja del cumpleaños", aunque en realidad no es una paradoja. La verdadera paradoja del cumpleaños es que la "paradoja del cumpleaños"no es una paradoja.

Demos un nombre a esta probabilidad:

$$\text{BirthdayProb}(q, N) \stackrel{\text{def}}{=} \Pr[\mathcal{B}(q, N) \text{ produce 1}].$$

Es posible escribir una fórmula exacta para esta probabilidad:

Lema 6.4.1. $\text{BirthdayProb}(q, N) = 1 - \prod_{i=1}^{q-1} \left(1 - \frac{i}{N}\right)$.

Demostración. En lugar de eso, calculemos la probabilidad de que $\mathcal{B}$ produzca 0, lo que nos permitirá luego resolver la probabilidad de que produzca 1. Para que $\mathcal{B}$ produzca 0, debe evitar las condiciones de terminación temprana en cada iteración del bucle principal. Por lo tanto:

$$\begin{aligned}\Pr[\mathcal{B}(q, N) \text{ produce 0}] = &\Pr[\mathcal{B}(q, N) \text{ no termina temprano en la iteración } i = 1] \\ &\cdot \Pr[\mathcal{B}(q, N) \text{ no termina temprano en la iteración } i = 2] \\ &\vdots \\ &\cdot \Pr[\mathcal{B}(q, N) \text{ no termina temprano en la iteración } i = q]\end{aligned}$$

En la iteración i del bucle principal, hay $i - 1$ valores previamente elegidos $s_1, ..., s_{i-1}$. El programa termina temprano si cualquiera de estos se elige nuevamente como s_i; de lo contrario, continúa con la siguiente iteración. Dicho de otra manera, hay $i - 1$ (de N) formas de elegir s_i que conducen a la terminación temprana; todas las demás elecciones de s_i evitan la terminación temprana. Dado que las N posibilidades para s_i ocurren con igual probabilidad:

$$\Pr[\mathcal{B}(q, N) \text{ no termina temprano en la iteración } i] = 1 - \frac{i-1}{N}.$$

Juntando todo:

$$\begin{aligned}\text{BirthdayProb}(q, N) &= \Pr[\mathcal{B}(q, N) \text{ produce 1}] \\ &= 1 - \Pr[\mathcal{B}(q, N) \text{ produce 0}] \\ &= 1 - (1 - \frac{1}{N})(1 - \frac{2}{N}) \cdots (1 - \frac{q-1}{N}) \\ &= 1 - \prod_{i=1}^{q-1}(1 - \frac{i}{N})\end{aligned}$$

Esto completa la demostración. ■

Ejemplo 6.6. Esta fórmula para $\text{BirthdayProb}(q, N)$ no es fácil de entender de un vistazo. Podemos obtener una mejor comprensión de su comportamiento como una función de q al graficarla. A continuación, se muestra un gráfico con $N = 365$, correspondiente al problema clásico del cumpleaños:

Con solo $q = 23$ personas, la probabilidad de un cumpleaños compartido ya supera el 50 %. El gráfico podría extenderse hacia la derecha (hasta $q = 365$), pero incluso con $q = 70$, la probabilidad supera el 99.9 %.

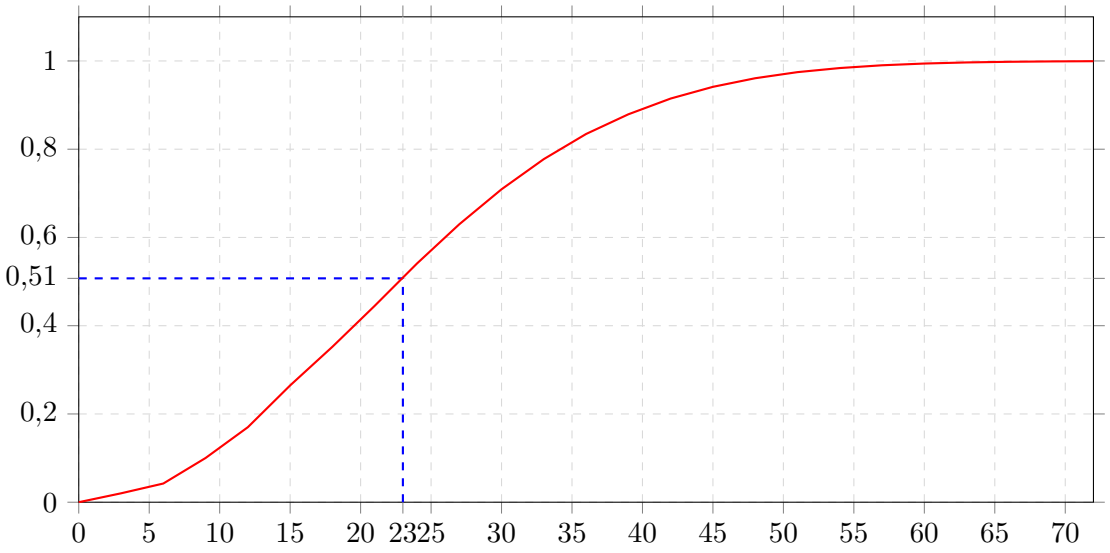


Figura 6.1: BirthdayProb$(q, 365)$

6.4.2. Acotaciones asintóticas en la probabilidad del cumpleaños

Será útil tener una fórmula *asintótica* de cómo crece BirthdayProb(q, N) como una función de q y N. Estamos más interesados en el caso donde q es relativamente pequeño comparado con N (p. ej., cuando q es una función polinómica de λ pero N es exponencial).

Lema 6.4.2. *Si* $q < \sqrt{2N}$, *entonces*

$$0{,}632\frac{q(q-1)}{2N} < \mathit{BirthdayProb}(q, N) < \frac{q(q-1)}{2N}$$

Dado que las acotaciones superior e inferior difieren solo por un factor constante, tiene sentido escribir $\mathrm{BirthdayProb}(q, N) = \Theta(q^2/N)$.

Demostración. Dividimos la demostración en dos partes. En esta sección, nos centraremos en la demostración del límite superior.

- Para demostrar la cota superior, usamos el hecho de que cuando x e y son números reales positivos, se cumple la siguiente desigualdad:

  $$(1-x)(1-y) = 1-(x+y)+xy \geq 1-(x+y).$$

  Esta desigualdad se debe a que xy es positivo y, por lo tanto, sumar xy a $1-(x+y)$ resulta en un valor mayor o igual.

  Esta idea se puede generalizar a un producto de múltiples términos. Si todos los términos x_i son positivos, entonces:

  $$\prod_{i=1}^{n}(1-x_i) \geq 1-\sum_{i=1}^{n}x_i.$$

  Esta generalización se puede demostrar por inducción.

A partir de la desigualdad anterior, se deduce la siguiente consecuencia, que será crucial para nuestra demostración:

$$1 - \prod_{i=1}^{n}(1 - x_i) \leq \sum_{i=1}^{n} x_i.$$

Esta se obtiene restando 1 a ambos lados y multiplicando por -1 (lo que invierte el signo de la desigualdad).

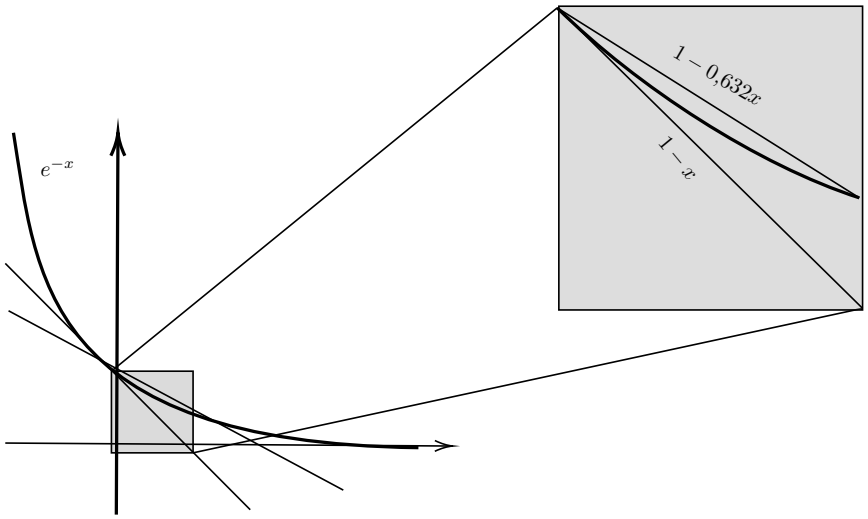
Ahora, aplicamos esta consecuencia a la fórmula de la probabilidad del cumpleaños:

$$\text{BirthdayProb}(q, N) \stackrel{\text{def}}{=} 1 - \prod_{i=1}^{q-1}\left(1 - \frac{i}{N}\right) \leq \sum_{i=1}^{q-1}\frac{i}{N} = \frac{1}{N}\sum_{i=1}^{q-1} i = \frac{q(q-1)}{2N}.$$

- Para demostrar el límite inferior, usamos el hecho de que cuando $0 \leq x \leq 1$,

$$1 - x \leq e^{-x} \leq 1 - 0{,}632x.$$

Este hecho se ilustra a continuación. El significado de 0.632 es que $1 - \frac{1}{e} = 0{,}63212\ldots$



Podemos usar ambos, los límites superior e inferior de e^{-x}, para demostrar lo siguiente:

$$\prod_{i=1}^{q-1}\left(1 - \frac{i}{N}\right) \leq \prod_{i=1}^{q-1} e^{-\frac{i}{N}} = e^{-\sum_{i=1}^{q-1}\frac{i}{N}} = e^{-\frac{q(q-1)}{2N}} \leq 1 - 0{,}632\frac{q(q-1)}{2N}.$$

Con la última desigualdad usamos el hecho de que $q \leq \sqrt{2N}$, y por lo tanto $\frac{q(q-1)}{2N} \leq 1$ (esto es necesario para aplicar la desigualdad $e^{-x} \leq 1 - 0{,}632x$). Por lo tanto:

$$\text{BirthdayProb}(q, N) \stackrel{\text{def}}{=} 1 - \prod_{i=1}^{q-1}\left(1 - \frac{i}{N}\right) \geq 1 - \left(1 - 0{,}632\frac{q(q-1)}{2N}\right) = 0{,}632\frac{q(q-1)}{2N}.$$

Esto completa la demostración.

■

Ejemplo 6.7. A continuación, se muestra una gráfica de estas acotaciones comparados con el valor real de $\text{BirthdayProb}(q, N)$ (para $N = 365$):

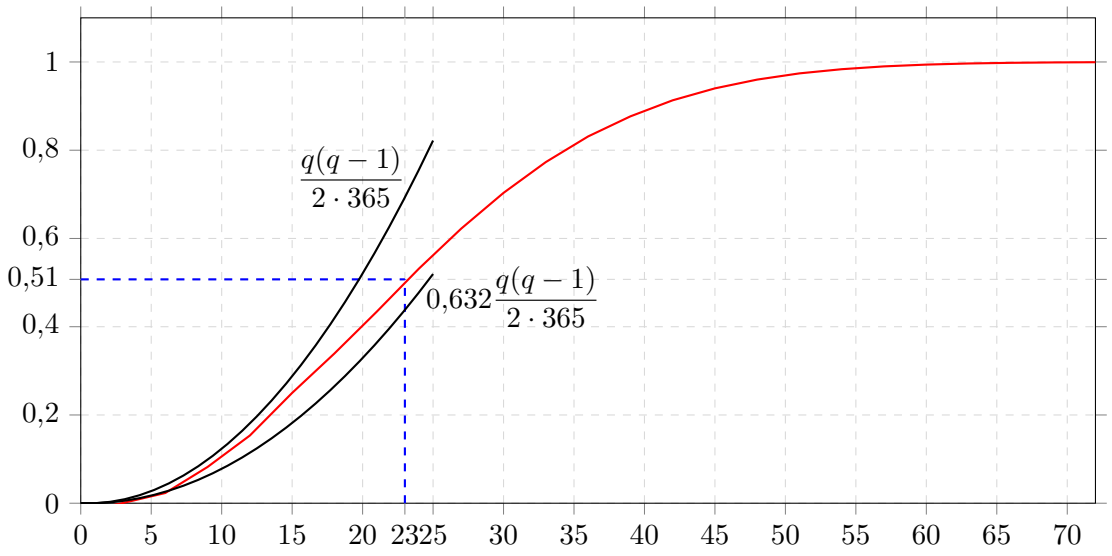


Figura 6.2: $\text{BirthdayProb}(q, 365)$

Como se mencionó anteriormente, $\text{BirthdayProb}(q, N)$ crece aproximadamente como q^2/N dentro del rango de valores que nos interesan (q pequeño en relación con N).

6.4.3. El problema del cumpleaños en términos de librerías indistinguibles

A continuación, se presentan dos librerías que también serán útiles para temas futuros.

$\mathcal{L}_{\text{samp-0}}$

SAMP():
$r \leftarrow \{0,1\}^\lambda$
return r

$\mathcal{L}_{\text{samp-1}}$

$R := \emptyset$

SAMP():
$r \leftarrow \{0,1\}^\lambda \setminus R$
$R := R \cup \{r\}$
return r

Ambas librerías proporcionan una subrutina SAMP que muestrea un elemento aleatorio de $\{0,1\}^\lambda$. La implementación en $\mathcal{L}_{\text{samp-0}}$ muestrea uniforme e independientemente de $\{0,1\}^\lambda$ cada vez. Muestrea **con reemplazo**, por lo que es posible (aunque poco probable) que múltiples llamadas a SAMP devuelvan el mismo valor en $\mathcal{L}_{\text{samp-0}}$.

Por otro lado, $\mathcal{L}_{\text{samp-1}}$ muestrea cadenas de λ bits **sin reemplazo**. Mantiene un registro de un conjunto R, que contiene todos los valores que ha muestreado previamente, y evita elegirlos de nuevo ("$\{0,1\}^\lambda \setminus R$" es

el conjunto de cadenas de λ bits excluyendo las que están en R). En esta librería, SAMP nunca devolverá el mismo valor dos veces.

La estrategia de distinción "obvia". Una forma natural (pero quizás no la única) de distinguir estas dos librerías, por lo tanto, sería llamar a SAMP muchas veces. Si alguna vez observas una salida repetida, entonces seguramente estás vinculado a $\mathcal{L}_{\text{samp-0}}$. Después de cierto número de llamadas a SAMP, si aún no ves ninguna salida repetida, podrías eventualmente detenerte y suponer que estás vinculado a $\mathcal{L}_{\text{samp-1}}$.

Sea $\mathcal{A}_q$ el programa de llamada "obvio" que realiza q llamadas a SAMP y devuelve 1 si ve un valor repetido. Claramente, el programa nunca puede devolver 1 cuando está vinculado a $\mathcal{L}_{\text{samp-1}}$. Por otro lado, cuando está vinculado a $\mathcal{L}_{\text{samp-0}}$, devuelve 1 con una probabilidad exactamente igual a $\text{BirthdayProb}(q, 2^\lambda)$. Por lo tanto, la ventaja de $\mathcal{A}_q$ es exactamente $\text{BirthdayProb}(q, 2^\lambda)$.

Este programa se comporta de manera diferente en presencia de estas dos librerías, por lo tanto, no son intercambiables. Pero, ¿son las bibliotecas indistinguibles? Hemos construido un programa de llamada con una ventaja de $\text{BirthdayProb}(q, 2^\lambda)$. No hemos especificado q exactamente, pero si $\mathcal{A}_q$ está destinado a ejecutarse en tiempo polinómico (como una función de λ), entonces q debe ser una función polinómica de λ. Entonces, la ventaja de $\mathcal{A}_q$ es $\text{BirthdayProb}(q, 2^\lambda) = \Theta(q^2/2^\lambda)$, ¡lo cual es despreciable!

Para mostrar que las liebrerías son indistinguibles, tenemos que mostrar que **todos** los programas tienen una ventaja despreciable. No es suficiente solo mostrar que este programa $\mathcal{A}_q$ en particular tiene una ventaja despreciable. ¡Quizás sorprendentemente, el programa de llamada "obvio" que hemos considerado es el **mejor diferenciador posible**!

Lema 6.4.3 (Muestreo con reemplazo)**.** *Sean $\mathcal{L}_{samp\text{-}0}$ y $\mathcal{L}_{samp\text{-}1}$ definidas como antes. Entonces, para todos los programas de llamada $\mathcal{A}$ que realizan q consultas a la subrutina SAMP, la ventaja de $\mathcal{A}$ al distinguir las librerías es* ***a lo sumo*** $\text{BirthdayProb}(q, 2^\lambda)$.

En particular, cuando $\mathcal{A}$ es de tiempo polinómico (en λ), q crece como un polinomio en el parámetro de seguridad. Por lo tanto, $\mathcal{A}$ tiene una ventaja ***despreciable****. Dado que esto es cierto para todo $\mathcal{A}$ de tiempo polinómico, tenemos $\mathcal{L}_{samp\text{-}0} \approx \mathcal{L}_{samp\text{-}1}$.*

Demostración. Considere las siguientes dos librerías:

$\mathcal{L}_{\text{hyb-0}}$

$R := \emptyset$
bad:= 0

SAMP():
$r \leftarrow \{0,1\}^\lambda$
if $r \in R$ then:
  bad:= 1

$R := R \cup \{r\}$
return r

$\mathcal{L}_{\text{hyb-1}}$

$R := \emptyset$
bad:= 0

SAMP():
$r \leftarrow \{0,1\}^\lambda$
if $r \in R$ then:
  bad:= 1
  $r \leftarrow \{0,1\}^\lambda \setminus R$
$R := R \cup \{r\}$
return r

$\mathcal{L}_{\text{samp-0}} = \mathcal{L}_{\text{hyb-0}}$: Observe que $\mathcal{L}_{\text{hyb-0}}$ simplemente muestrea uniformemente de $\{0,1\}^\lambda$. Las variables adicionales R y bad en $\mathcal{L}_{\text{hyb-0}}$ en realidad no tienen ningún efecto en su comportamiento externo (se usan

solo por conveniencia más adelante en la prueba).

$\mathcal{L}_{\text{hyb-1}} = \mathcal{L}_{\text{samp-1}}$: Mientras que $\mathcal{L}_{\text{samp-1}}$ evita repeticiones simplemente muestreando de $\{0,1\}^\lambda \setminus R$, esta librería $\mathcal{L}_{\text{hyb-1}}$ muestrea r uniformemente de $\{0,1\}^\lambda$ y vuelve a intentarlo si el resultado está en R. Este método se llama **muestreo por rechazo**, y tiene el mismo efecto que muestrear r directamente de $\{0,1\}^\lambda \setminus R$.

Convenientemente, $\mathcal{L}_{\text{hyb-0}}$ y $\mathcal{L}_{\text{hyb-1}}$ difieren solo en el código al que se llega cuando bad $= 1$ (resaltado). Por lo tanto, usando el Lema 6.3.3, podemos acotar la ventaja del programa de llamada:

$$\begin{aligned}
&|\Pr[\mathcal{A} \diamond L_{\text{samp-L}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond L_{\text{samp-R}} \Rightarrow 1]| \\
&= |\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-0}} \Rightarrow 1] - \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-1}} \Rightarrow 1]| \\
&\leq \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{hyb-0}} \text{ establece bad}:= 1].
\end{aligned}$$

Finalmente, podemos observar que $\mathcal{A} \diamond \mathcal{L}_{\text{hyb-0}}$ establece bad$:= 1$ solo en el caso de que vea una muestra repetida de $\{0,1\}^\lambda$. Esto ocurre con probabilidad $\text{BirthdayProb}(q, 2^\lambda)$.

■

Discusión

- Expresar el problema del cumpleaños en términos de librerías indistinguibles lo convierte en una herramienta útil en futuras pruebas de seguridad. Por ejemplo, al demostrar la seguridad de una construcción, podemos reemplazar un paso de muestreo uniforme con un paso de muestreo sin reemplazo. Este cambio tiene solo un efecto insignificante, pero ahora el resto de la prueba puede aprovechar el hecho de que las muestras nunca se repiten.

  Otra forma de decir esto es que, cuando se piensa en una construcción criptográfica, es "seguro asumir" que las cadenas largas muestreadas aleatoriamente no se repiten y se comportan en consecuencia.

- Sin embargo, si una prueba de seguridad utiliza la indistinguibilidad de las bibliotecas del cumpleaños, significa que es probable que el esquema se rompa cuando un usuario repite un valor muestreado uniformemente. Dado que esto se vuelve inevitable a medida que el número de muestras se acerca a $\sqrt{2^{\lambda+1}} \sim 2^{\lambda/2}$, significa que el esquema solo ofrece $\lambda/2$ bits de seguridad. Cuando un esquema tiene esta propiedad, decimos que tiene **seguridad limitada por el cumpleaños** (birthday bound security). Es importante comprender cuándo un esquema tiene esta propiedad, ya que informa el tamaño de las claves que se deben elegir en la práctica.

Capítulo 7

Generadores pseudoaleatorios

El one-time pad requiere una clave que sea tan larga como el texto plano. Olvidemos que conocemos esta limitación. Supongamos que Alice y Bob comparten solo una clave secreta corta de λ bits, pero quieren cifrar un texto plano m de 2λ bits. No saben que la secrecía perfecta del one-time pad es imposible en esta configuración, por lo que intentan que funcione de todos modos usando el siguiente razonamiento:

- El único esquema de cifrado que conocen es el one-time pad, por lo que deciden que el texto cifrado tendrá la forma $c = m \oplus \texttt{??}$. Esto significa que el valor desconocido `??` debe tener una longitud de 2λ bits.
- Para que se aplique la seguridad del one-time pad, el valor desconocido `??` debe distribuirse uniformemente.
- El proceso de obtener el valor desconocido `??` de la clave compartida k debe ser determinista, para que el remitente y el receptor calculen el mismo valor y el descifrado funcione correctamente.

Sea G el proceso que transforma la clave k en este valor misterioso. Entonces $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$, y el esquema de cifrado es $\text{Enc}(k, m) = m \oplus G(k)$.

No es difícil ver que si G es una función determinista, entonces solo hay 2^λ posibles salidas de G, por lo que la distribución de $G(k)$ no puede ser uniforme en $\{0,1\}^{2\lambda}$. Por lo tanto, no podemos argumentar que el esquema sea seguro de la misma manera que el one-time pad.

Sin embargo, ¿qué sucede si la distribución de los valores de $G(k)$ no es perfectamente uniforme, sino solo "lo suficientemente cercana" a la uniformidad? Supongamos que ningún algoritmo de tiempo polinómico puede distinguir la distribución de los valores de $G(k)$ de la distribución uniforme. Entonces, seguramente esto debería ser "lo suficientemente cercano" a la uniformidad para fines prácticos. Esta es exactamente la idea de la **pseudoaleatoriedad**. Resulta que si G tiene una propiedad de pseudoaleatoriedad, entonces el esquema de cifrado descrito anteriormente es realmente seguro (contra adversarios de tiempo polinómico, en el sentido discutido en el capítulo anterior).

7.1. Definiciones

Un **generador pseudoaleatorio (PRG)** es una función determinista G cuyas salidas son más largas que sus entradas. Cuando la entrada a G se elige uniformemente al azar, induce una cierta distribución sobre la posible salida. Como se discutió anteriormente, esta distribución de salida **no puede ser uniforme**. Sin embargo, la distribución es *pseudoaleatoria* si es **indistinguible de la distribución uniforme**. Más formalmente:

Definición 7.1.1. *Sea $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell}$ una función determinista con $\ell > 0$. Decimos que G es un **generador pseudoaleatorio (PRG) seguro** (seguridad del PRG) si $\mathcal{L}^G_{prg} \approx \mathcal{L}^G_{prg\text{-}\$}$, donde:*

$\mathcal{L}^G_{prg}$

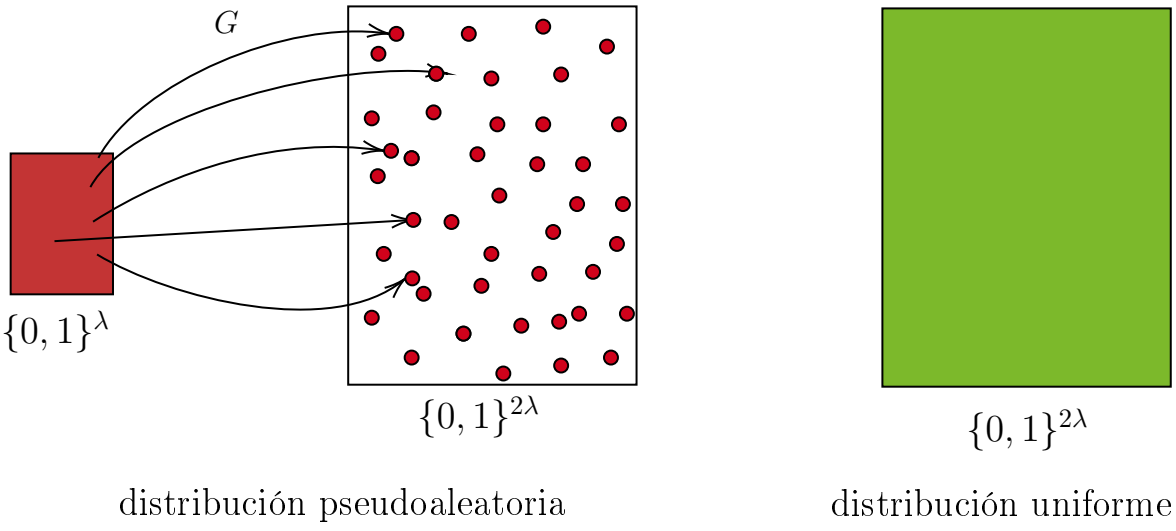
QUERY():
$s \leftarrow \{0,1\}^\lambda$
return $G(s)$

$\mathcal{L}^G_{prg\text{-}\$}$

QUERY():
$r \leftarrow \{0,1\}^{\lambda+\ell}$
return r

*El valor ℓ se llama el **factor de expansión** del PRG. La entrada al PRG se llama típicamente una **semilla**.*

A continuación, se muestra una ilustración de las distribuciones muestreadas por estas librerías, para un PRG que duplica la longitud ($\ell = \lambda$) (no dibujado a escala):



La librería $\mathcal{L}_{\text{prg}}$ toma muestras de la distribución de puntos rojos, primero muestreando un elemento uniforme de $\{0,1\}^\lambda$ y realizando la acción de G en ese valor para obtener un resultado rojo en $\{0,1\}^{2\lambda}$. La otra librería $\mathcal{L}_{\text{prg-\$}}$ muestrea directamente la distribución uniforme en $\{0,1\}^{2\lambda}$ (en verde arriba).

Para comprender los PRG, debes apreciar simultáneamente dos formas de comparar la distribución de salida del PRG con la distribución uniforme:

- **Desde una perspectiva relativa**, la distribución de salida del PRG es diminuta. De las $2^{2\lambda}$ cadenas en $\{0,1\}^{2\lambda}$, solo 2^λ son salidas posibles de G. Estas cadenas representan una fracción $2^\lambda/2^{2\lambda} = 1/2^\lambda$ de $\{0,1\}^{2\lambda}$: ¡una fracción despreciable!

- **Desde una perspectiva absoluta**, la distribución de salida del PRG es enorme. Hay 2^λ salidas posibles de G, ¡lo cual es una cantidad exponencial!

La ilustración anterior solo captura la perspectiva relativa (comparando los puntos rojos con la extensión total de $\{0,1\}^{2\lambda}$), por lo que puede dar lugar a cierta confusión. Con solo mirar esta imagen, es difícil imaginar cómo las dos distribuciones podrían ser indistinguibles. ¿Cómo podría un programa de llamada no notar si está viendo el conjunto completo o solo una fracción despreciable del conjunto completo? Bueno, si se ejecuta en tiempo polinómico en λ, entonces 2^λ y $2^{2\lambda}$ son ambos tan enormes que realmente no importa que uno sea mucho mayor que el otro. Los tamaños relativos de la distribución realmente no ayudan a distinguir, ya que no es una estrategia viable para el distinguidor "medir" el tamaño de la distribución que está muestreando.

Consideremos: hay alrededor de 2^{75} moléculas en una cucharadita de agua, y alrededor de 2^{275} moléculas de agua en los océanos de la Tierra. Supongamos que viertes una cucharadita de agua en el océano y dejas que se mezcle durante unos miles de años. Aunque la cucharadita representa solo $1/2^{200}$ del contenido de los océanos, ¡eso no facilita el seguimiento de las 2^{75} moléculas de agua que se originaron en la cucharadita! Si eres lo suficientemente pequeño como para ver moléculas de agua individuales, entonces una cucharadita de agua se ve tan grande como el océano.

Discusión y Trampas

- No confundas la interfaz de un PRG (que toma una semilla como entrada) con la interfaz de las librerías de seguridad $\mathcal{L}^G_{\text{prg}}$ y $\mathcal{L}^G_{\text{prg-\$}}$ (¡su subrutina QUERY no toma ninguna entrada!). Un PRG es, de hecho, un algoritmo al que puedes alimentar cualquier cadena que desees. Sin embargo, la seguridad solo está garantizada cuando el PRG se utiliza exactamente como se describe en las librerías de seguridad, en particular, cuando la semilla se elige de manera uniforme/secreta y no se utiliza para nada más.

  Nada impide que un usuario ponga una s elegida por un adversario en un PRG, o revele una semilla de PRG a un adversario, etc. Simplemente no obtienes ninguna garantía de seguridad al hacerlo, ya que no es la situación reflejada en las librerías de seguridad del PRG.

- Realmente no tiene sentido decir que "0010110110 es una cadena aleatoria" o "0000000001 es una cadena pseudoaleatoria". La aleatoriedad y la pseudoaleatoriedad son propiedades del **proceso** utilizado para generar una cadena, no propiedades de las cadenas individuales en sí mismas. Cuando tenemos un valor $z = G(s)$ donde G es un PRG y s se elige uniformemente, se podría decir que z fue "elegido pseudoaleatoriamente". Se podría decir que la salida de algún proceso es una "distribución pseudoaleatoria". Pero es un poco descuidado (aunque común) decir que una cadena z "es pseudoaleatoria".

- Existen pruebas estadísticas comunes que puedes ejecutar, que verifican si algunos datos tienen varias propiedades que esperarías de una distribución uniforme[1]. Por ejemplo, ¿hay aproximadamente el mismo número de 0s y 1s? ¿La subcadena `01010` aparece con aproximadamente la frecuencia que esperaría? Si interpreto la cadena como una serie de puntos en el cuadrado unitario $[0,1)^2$, ¿es cierto que aproximadamente $\pi/4$ de ellos están a una distancia euclidiana 1 del origen?

La definición de pseudoaleatoriedad es una especie de definición "maestra" que abarca todas estas pruebas estadísticas y más. Después de todo, ¿qué es una prueba estadística, sino un procedimiento de tiempo

[1] Para una lista de tales pruebas, consulta http://csrc.nist.gov/publications/nistpubs/800-22-rev1a/SP800-22rev1a.pdf.

polinómico que obtiene muestras de una distribución y produce una decisión de sí/no? La pseudoaleatoriedad significa que cada prueba estadística que "pasa" datos uniformes también "pasar" datos generados pseudoaleatoriamente.

7.2. Generadores Pseudoaleatorios en la práctica

Probablemente esperas ver al menos un ejemplo de un PRG seguro. Desafortunadamente, las cosas no son tan simples. ¡No tenemos ejemplos de PRG seguros! Si fuera posible probar que alguna función G es un PRG seguro, resolvería el famoso problema **P vs NP**, el problema sin resolver más famoso en la informática (y podría decirse que en todas las matemáticas).

Lo mejor que la investigación criptográfica puede ofrecer son **PRGs candidatos**, que se *conjeturan* que son seguros. Los mejores ejemplos de tales PRGs son aquellos que han sido sometidos a un escrutinio público significativo y han resistido todos los intentos de ataques hasta ahora.

De hecho, el resto de este curso se basa en criptografía que solo se *conjetura* que es segura. ¿Cómo es esto posible, dado el enfoque declarado de la asignatura en la seguridad demostrable? A medida que avances en el curso, presta atención a cómo todas las afirmaciones de seguridad demostrable son **condicionales**: *si X es seguro, entonces Y es seguro*. Podrás rastrear a través de esta red de implicaciones y descubrir que solo hay un pequeño número de primitivas criptográficas subyacentes cuya seguridad es meramente conjeturada (los PRGs son un ejemplo de tal primitiva). Todo lo demás se construye sobre estas primitivas de una manera demostrablemente segura.

Con esa exención de responsabilidad, seguramente ahora se te puede mostrar un ejemplo de un PRG conjeturado seguro, ¿verdad? De hecho, hay algunos PRGs conjeturados que son lo suficientemente simples como para mostrarte en este punto, pero no los encontrarás en este manuscrito. El problema es que ninguno de estos PRGs candidatos se utiliza realmente en la práctica. Cuando realmente necesitas un PRG en la práctica, en realidad usarías un PRG que se construye a partir de algo llamado un **cifrado en bloque** (que no veremos hasta el siguiente capítulo). Un cifrado en bloque es conceptualmente más complicado que un PRG, e incluso se puede construir a partir de un PRG (en principio). Eso explica por qué este curso comienza con los PRGs. En la práctica, un cifrado en bloque es simplemente un objeto más útil, por lo que es lo que encontrarías fácilmente disponible (incluso implementado con instrucciones especializadas de la CPU en la mayoría de las CPU).

7.2.1. Cómo NO construir un PRG

Podemos apreciar los desafíos involucrados en la construcción de un PRG "desde cero" al observar primero una idea obvia para un PRG y comprender por qué es inseguro.

Ejemplo 7.1. Centrémonos en el caso de un PRG que duplica la longitud. Debe tomar λ bits como entrada y generar 2λ bits como salida. La salida debería parecer aleatoria cuando la entrada se muestrea uniformemente. Una idea natural es que el PRG candidato simplemente repita la entrada dos veces. Después de todo, si la entrada s es aleatoria, entonces $s||s$ también es aleatoria, ¿verdad?

$\underline{G(s)}$:
return $s||s$

Para entender por qué este PRG es inseguro, primero pregúntate si las siguientes cadenas parecen haber sido muestreadas uniformemente de $\{0,1\}^8$:

`11011101`, `01010101`, `01110111`, `01000100`, ...

¿Ves algún patrón? Cada cadena tiene su primera mitad igual a su segunda mitad. Ese es un patrón conspicuo porque es relativamente raro que una cadena elegida uniformemente tenga esta propiedad.

Por supuesto, esto es exactamente lo que está mal con este PRG simplista G definido anteriormente. Cada salida de G tiene la primera/segunda mitad iguales. Pero es raro que las cadenas muestreadas uniformemente tengan esta propiedad. Podemos formalizar esta observación como un ataque contra la seguridad PRG de G:

$\mathcal{A}$
$x||y :=$QUERY()
return $x \stackrel{?}{=} y$

La primera línea significa obtener el resultado de QUERY y establecer su primera mitad para que sea la cadena x y su segunda mitad para que sea y. Este programa de llamada simplemente verifica si la salida de QUERY tiene mitades iguales.

Para completar el ataque, debemos mostrar que este programa de llamada tiene un sesgo no despreciable que distingue las librerías $\mathcal{L}_{\text{prg}}$ y $\mathcal{L}_{\text{prg-\$}}$.

- Cuando se vincula a $\mathcal{L}_{\text{prg}}$, el programa de llamada recibe salidas de G, que siempre tienen la primera/segunda mitad coincidentes. Entonces $\Pr[\mathcal{A} \diamond \mathcal{L}^G_{\text{prg}} \Rightarrow 1] = 1$. A continuación, hemos completado $\mathcal{L}_{\text{prg}}$ con los detalles de nuestro algoritmo G:

$\mathcal{A}$
$x||y :=$QUERY()
return $x \stackrel{?}{=} y$

$\diamond$

$\mathcal{L}^G_{\text{prg}}$
QUERY():
$s \leftarrow \{0,1\}^\lambda$
return $s||s$

- Cuando se vincula a $\mathcal{L}_{\text{prg-rand}}$, el programa de llamada recibe muestras uniformes de $\{0,1\}^{2\lambda}$.

$\mathcal{A}$
$x||y :=$QUERY()
return $x \stackrel{?}{=} y$

$\diamond$

$\mathcal{L}^G_{\text{prg-\$}}$
QUERY():
$r \leftarrow \{0,1\}^{2\lambda}$
return r

  $\mathcal{A}$ produce 1 siempre que muestreamos una cadena de $\{0,1\}^{2\lambda}$ con la primera/segunda mitad iguales. ¿Cuál es exactamente la probabilidad de que esto ocurra? Hay varias maneras de ver que la probabilidad es $1/2^\lambda$ (esto es como preguntar la probabilidad de sacar dobles con dos dados, pero cada dado tiene 2^λ caras en lugar de 6). Por lo tanto, $\Pr[\mathcal{A} \diamond \mathcal{L}^G_{\text{prg-rand}} \Rightarrow 1] = 1/2^\lambda$.

La ventaja de este adversario es $1 - 1/2^\lambda$, que ciertamente no es despreciable, ni siquiera se acerca a 0 a medida que λ crece. Esto demuestra que G no es un PRG seguro.

Este ejemplo ilustra cómo la aleatoriedad/pseudoaleatoriedad es una propiedad de todo el *proceso*, no de cadenas individuales. Si tomas una cadena de 1s y la concatenas con otra cadena de 1s, obtienes una cadena larga de 1s. "Contener solo 1s" es una propiedad de las cadenas individuales. Si tomas una "cadena aleatoria" y la concatenas con otra "cadena aleatoria", es posible que no obtengas una "cadena larga aleatoria". Ser aleatorio no es una propiedad de una cadena individual, sino de todo el proceso que la genera.

Las salidas de este G tienen la primera/segunda mitad iguales, lo cual es un patrón obvio. ¡El desafío de diseñar un PRG seguro es que sus salidas no deben tener ningún patrón discernible! Cualquier patrón conducirá a un ataque similar al que se muestra arriba.

7.2.2. Concepto relacionado: generación de números aleatorios

La seguridad de un PRG requiere que la semilla se elija uniformemente. En la práctica, la semilla tiene que provenir de alguna parte. Generalmente, una fuente de "aleatoriedad" es proporcionada por el hardware o el sistema operativo, y el proceso que genera estos bits aleatorios se llama (confusamente) un *generador de números aleatorios (RNG)*.

En este curso no cubriremos la generación de números aleatorios de bajo nivel, sino que simplemente señalaremos en qué se diferencia de los PRG que estudiamos:

- El trabajo de un PRG es tomar una pequeña cantidad de aleatoriedad "ideal" (en otras palabras, uniforme) y extenderla.
- Por el contrario, un RNG generalmente toma muchas entradas a lo largo del tiempo y mantiene un estado interno. Estas entradas a menudo provienen de fuentes físicas/de hardware. Si bien estas entradas son "ruidosas" en cierto sentido, es difícil imaginar que sean estadísticamente uniformes. Entonces, el trabajo del RNG es "refinar" (a veces muchas) fuentes de datos ruidosos en salidas uniformes.

7.3. Aplicación: claves más cortas en el one-time pad

Revisamos el ejemplo motivador del comienzo de este capítulo. Alice y Bob comparten solo una clave de λ bits pero quieren cifrar un mensaje de longitud $\lambda+\ell$. La idea principal es expandir la clave k a una cadena más larga usando un PRG G, y usar el resultado como un one-time pad en el texto plano (más largo). Más precisamente, sea $G : \{0,1\}^\lambda \to \{0,1\}^{\lambda+\ell}$ un PRG, y definamos el siguiente esquema de cifrado:

pseudo-OTP			
$\mathcal{K} = \{0,1\}^\lambda$	Gen:	Enc(k, m):	Dec(k, c):
$\mathcal{M} = \{0,1\}^{\lambda+\ell}$	$k \leftarrow \{0,1\}^\lambda$	return $G(k) \oplus m$	return $G(k) \oplus c$
$\mathcal{C} = \{0,1\}^{\lambda+\ell}$	return k		

El esquema resultante no tendrá la seguridad perfecta del one-time pad. Es decir, los cifrados de m_0 y m_1 no se distribuirán idénticamente en general. Sin embargo, las distribuciones serán indistinguibles si G es un PRG seguro. El matiz preciso de seguridad obtenido por esta construcción es el siguiente.

Definición 7.3.1. *Sea Σ un esquema de cifrado. Entonces Σ tiene indistinguibilidad (computacional) de un solo uso*

$\mathcal{L}^{\Sigma}_{otind\text{-}0}$

$EAVESDROP(m_0, m_1 \in \Sigma.\mathcal{M})$:
$k \leftarrow \Sigma.Gen$
$c := \Sigma.Enc(k, m_0)$
return c

$\approx$

$\mathcal{L}^{\Sigma}_{otind\text{-}1}$

$EAVESDROP(m_0, m_1 \in \Sigma.\mathcal{M})$:
$k \leftarrow \Sigma.Gen$
$c := \Sigma.Enc(k, m_1)$
return c

Es decir, si para todos los distinguidores de tiempo polinómico $\mathcal{A}$, tenemos

$$\Pr[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{otind\text{-}0} \Rightarrow 1] \approx \Pr[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{otind\text{-}1} \Rightarrow 1].$$

Esto es esencialmente lo mismo que la Definición 5.4.2, excepto que estamos usando $\approx$ (indistinguibilidad) en lugar de $\equiv$ (intercambiabilidad).

Afirmación 7.3.2. *Sea pOTP el esquema de cifrado pseudo-OTP. Si pOTP se instancia usando un PRG seguro G, entonces pOTP tiene indistinguibilidad computacional de un solo uso.*

Demostración. Debemos mostrar que $\mathcal{L}^{\text{pOTP}}_{\text{otind-0}} \approx \mathcal{L}^{\text{pOTP}}_{\text{otind-1}}$. Como de costumbre, procederemos usando una secuencia de librerías que comienza en $\mathcal{L}^{\text{pOTP}}_{\text{otind-0}}$ y termina en $\mathcal{L}^{\text{pOTP}}_{\text{otind-1}}$. Para cada librería intermedia, demostraremos que es indistinguible de la anterior. Ten en cuenta que se nos permite usar el hecho de que G es un PRG seguro. En términos prácticos, esto significa que si podemos expresar alguna librería intermedia en términos de $\mathcal{L}^{G}_{\text{prg}}$ (una de las librerías en la definición de seguridad del PRG), podemos reemplazarla con su contraparte $\mathcal{L}^{G}_{\text{prg-\$}}$ (o viceversa). La seguridad PRG de G dice que tal cambio será indistinguible.

$\mathcal{L}^{\text{pOTP}}_{\text{otind-0}}$:

$\mathcal{L}^{\text{pOTP}}_{\text{otind-0}}$

EAVESDROP$(m_0, m_1 \in \{0,1\}^{\lambda+\ell})$:
$k \leftarrow \{0,1\}^{\lambda}$
$c := G(k) \oplus m_0$
return c

El punto de partida es $\mathcal{L}^{\text{pOTP}}_{\text{otind-0}}$, que se muestra aquí con los detalles de pOTP completos.

$\mathcal{L}_{\text{hyb-1}}$:

EAVESDROP(m_0, m_1):
$z \leftarrow$ QUERY()
$c := z \oplus m_0$
return c

$\diamond$

$\mathcal{L}^{G}_{\text{prg}}$

QUERY():
$s \leftarrow \{0,1\}^{\lambda}$
return $G(s)$

El primer paso intermedio es factorizar los cálculos que involucran a G, en términos de la librería $\mathcal{L}^{G}_{\text{prg}}$.

$\mathcal{L}_{\text{hyb-2}}$:

EAVESDROP(m_0, m_1):
$z \leftarrow$ QUERY()
$c := z \oplus m_0$
return c

$\diamond$

$\mathcal{L}^{G}_{\text{prg-\$}}$

QUERY():
$r \leftarrow \{0,1\}^{\lambda+\ell}$
return r

A partir de la seguridad PRG de G, podemos reemplazar la instancia de $\mathcal{L}^{G}_{\text{prg}}$ con $\mathcal{L}^{G}_{\text{prg-\$}}$. La librería resultante $\mathcal{L}_{\text{hyb-2}}$ es indistinguible de la anterior.

$\mathcal{L}_{\text{hyb-3}}$:

$\mathcal{L}^{\text{OTP}}_{\text{otint-\$}}$

EAVESDROP(m_0, m_1):

$z \leftarrow \{0,1\}^{\lambda+\ell}$

$c := z \oplus m_0$

return c

Se ha insertado una subrutina. Observe que la librería resultante es precisamente $\mathcal{L}^{\text{OTP}}_{\text{otint-0}}$ que involucra un **one-time pad estándar** en textos planos de tamaño $\lambda+\ell$. Hemos probado esencialmente que pOTP es indistinguible del OTP estándar y, por lo tanto, podemos aplicar la seguridad del OTP.

$\mathcal{L}_{\text{hyb-4}}$:

$\mathcal{L}^{\text{OTP}}_{\text{otint-\$}}$

EAVESDROP(m_0, m_1):

$z \leftarrow \{0,1\}^{\lambda+\ell}$

$c := z \oplus m_1$

return c

La intecambiabilidad de un solo uso del OTP nos permite reemplazar $\mathcal{L}^{\text{OTP}}_{\text{otint-0}}$ con $\mathcal{L}^{\text{OTP}}_{\text{otint-1}}$; son intercambiables.

El resto de la prueba es esencialmente un "reflejo" de los pasos anteriores, en los que realizamos los mismos pasos pero a la inversa (y con m_1 en lugar de m_0).

$\mathcal{L}_{\text{hyb-5}}$:

EAVESDROP(m_0, m_1):

$z \leftarrow$ QUERY()

$c := z \oplus m_1$

return c

$\diamond$

$\mathcal{L}^{G}_{\text{prg-\$}}$

QUERY():

$r \leftarrow \{0,1\}^{\lambda+\ell}$

return r

Se ha factorizado una declaración en una subrutina, que resulta coincidir exactamente con $\mathcal{L}^{G}_{\text{prg-\$}}$.

$\mathcal{L}_{\text{hyb-6}}$:

EAVESDROP(m_0, m_1):

$z \leftarrow$ QUERY()

$c := z \oplus m_1$

return c

$\diamond$

$\mathcal{L}^{G}_{\text{prg}}$

QUERY():

$s \leftarrow \{0,1\}^{\lambda}$

return $G(s)$

A partir de la seguridad del PRG de G, podemos reemplazar $\mathcal{L}^{G}_{\text{prg-\$}}$ con $\mathcal{L}^{G}_{\text{prg}}$. La librería resultante es indistinguible de la anterior.

$\mathcal{L}^{\text{pOTP}}_{\text{otind-1}}$:

$\mathcal{L}^{\text{pOTP}}_{\text{otind-1}}$

EAVESDROP(m_0, m_1):

$k \leftarrow \{0,1\}^{\lambda}$

$c := G(k) \oplus m_1$

return c

Se ha insertado una subrutina. El resultado es $\mathcal{L}^{\text{pOTP}}_{\text{otind-1}}$.

Resumiendo, mostramos una secuencia de librerías que satisfacen lo siguiente:

$$\mathcal{L}^{\text{pOTP}}_{\text{otind-0}} \equiv \mathcal{L}_{\text{hyb-1}} \approx \mathcal{L}_{\text{hyb-2}} \equiv \mathcal{L}_{\text{hyb-3}} \equiv \mathcal{L}_{\text{hyb-4}} \equiv \mathcal{L}_{\text{hyb-5}} \approx \mathcal{L}_{\text{hyb-6}} \equiv \mathcal{L}^{\text{pOTP}}_{\text{otind-1}}.$$

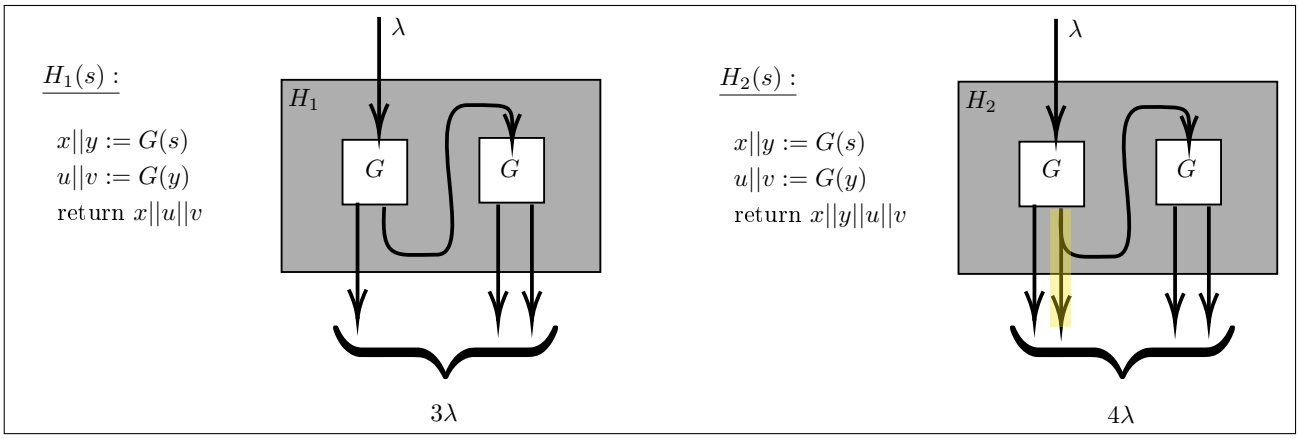
Por lo tanto, $\mathcal{L}^{\text{pOTP}}_{\text{otind-0}} \approx \mathcal{L}^{\text{pOTP}}_{\text{otind-1}}$, y pOTP tiene indistinguibilidad computacional de un solo uso. ■

7.4. Extendiendo la expansión de un PRG

El *factor de expansión* de un PRG mide cuánto más larga es su salida que su entrada. ¿Podemos usar un PRG con un factor de expansión pequeño para construir un PRG con un factor de expansión mayor? La respuesta es sí, ¡pero solo si lo haces de la manera correcta!

7.4.1. Dos enfoques para aumentar el estiramiento

Supongamos que $G : \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ es un PRG que duplica la longitud (es decir, un PRG con factor de expansión λ). A continuación, se presentan dos ideas para construir un PRG con un factor de expansión mayor:



Aunque las construcciones son similares, solo una de ellas es segura. Antes de seguir leyendo, ¿puedes adivinar cuál de H_1, H_2 es un PRG seguro y cuál es inseguro? Al comparar cuidadosamente estos dos enfoques, espero que desarrolles una mejor comprensión de la definición de seguridad de PRG.

7.4.2. Una prueba de seguridad

Creo que es útil ilustrar la "estrategia" de las pruebas de seguridad comenzando desde la conclusión deseada y trabajando hacia atrás. ¿Qué mejor manera de hacer esto que como un diálogo socrático al estilo de Galileo?

Salviati: Estoy seguro de que H_1 es el PRG seguro.

Simplicio: Si entiendo correctamente la definición de seguridad para los PRGs, quieres decir que la salida de H_1 parece indistinguible de una salida escogida de forma uniforme, cuando la entrada a H_1 es escogida de manera uniforme. ¿Por qué dices eso?

Salviati: ¡Simple! La salida de H_1 consiste en segmentos llamados x, u y v. Cada uno de estos son salidas de G, y dado que G en sí mismo es un PRG, sus salidas parecen uniformes.

SIMPLICIO: Ojalá tuviera tu audacia, Salvati. Yo mismo soy más cauteloso. Si G es un PRG seguro, entonces sus salidas son, de hecho, indistinguibles de uniforme, ¡pero seguramente solo cuando su entrada es uniforme! ¿Estás tan seguro de que este es el caso aquí?

SALVIATI: Haces una buena pregunta, Simplicio. En estos empeños, siempre es preferible pecar de cauteloso. Cuando queremos afirmar que H_1 es un PRG seguro, consideramos la naturaleza de sus salidas cuando su semilla s es uniforme. Dado que H_1 envía esa semilla s directamente a G, tu preocupación está resuelta.

SIMPLICIO: Sí, puedo ver cómo en la expresión $x\|y := G(s)$ la entrada a G es uniforme, y por lo que sus salidas x e y son indistinguibles de aleatorio. Dado que x es parte de la salida de H_1, ¡estamos progresando hacia mostrar que la salida completa de H_1 es indistinguible de aleatorio! Sin embargo, la salida de H_1 también contiene los términos u y v. Cuando examino cómo se generan, como $u\|v := G(y)$, me vuelvo a preocupar. Seguramente y no es uniforme, ¡así que no veo forma de aplicar la seguridad de G!

SALVIATI: ¡La respuesta no podría ser más obvia! Es cierto que y no está distribuido uniformemente. ¿Pero no te habías convencido ya de que y es indistinguible de uniforme? ¿Debería ser suficiente con eso?

SIMPLICIO: ¡Increíble! Creo que lo entiendo ahora. Permíteme resumir: Suponemos que la entrada s a H_1 se elige uniformemente y examinamos qué sucede con las salidas de H_1. En la expresión $x\|y := G(s)$, la entrada a G es uniforme y, por lo tanto, x e y son indistinguibles de uniforme. Ahora, considerando la expresión $u\|v := G(y)$, el resultado es indistinguible de un escenario en el que y es verdaderamente uniforme. ¡Pero si y fuera verdaderamente uniforme, esas salidas u y v serían indistinguibles de uniforme! En conjunto, x, u y v (las salidas de H_1) son cada una indistinguibles de uniforme.

Espero que esto hayas disfrutado del diálogo.[2] La prueba formal de seguridad y su secuencia de librerías seguirán el esquema dado en el resumen de Simplicio. Comenzamos aplicando la definición de seguridad de PRG a la primera llamada a G y reemplazamos sus salidas con valores verdaderamente uniformes. Después de este cambio, la entrada a la segunda llamada a G se vuelve uniforme, lo que nos permite aplicar la definición de seguridad de PRG nuevamente.

Afirmación 7.4.1. *Si G es un PRG seguro que duplica la longitud, entonces H_1 (definido anteriormente) es un PRG seguro (que triplica la longitud).*

Demostración. Uno de los aspectos más complicados de esta prueba es que estamos usando un PRG seguro G para probar la seguridad de otro PRG H_1. Eso significa que tanto $\mathcal{L}^{H_1}_{\text{prg(-\$)}}$ como $\mathcal{L}^{G}_{\text{prg(-\$)}}$ aparecerán en esta prueba. Ambas librerías/interfaces tienen una subrutina llamada "QUERY", y renombraremos estas subrutinas QUERY_{H_1} y QUERY_{G} para disambiguar.

Queremos mostrar que $\mathcal{L}^{H_1}_{\text{prg}} \approx \mathcal{L}^{H_1}_{\text{prg-\$}}$. Como de costumbre, lo hacemos con una secuencia de librerías. Dado que asumimos que G es un PRG seguro, se nos permite usar el hecho de que $\mathcal{L}^{G}_{\text{prg}} \approx \mathcal{L}^{G}_{\text{prg-\$}}$.

[2] Si te estás preguntando qué demonios acaba de pasar: en el libro de Galileo de 1632, *Diálogo sobre los dos máximos sistemas del mundo*, se presentan los argumentos a favor del heliocentrismo mediante un diálogo entre Salviati (quien defendía el modelo heliocéntrico) y Simplicio (quien creía en el modelo geocéntrico).

$\mathcal{L}^{H_1}_{\text{prg}}$:

$\mathcal{L}^{H_1}_{\text{prg}}$

QUERY$_{H_1}$():
$s \leftarrow \{0,1\}^\lambda$
$x||y := G(s)$
$u||v := G(y)$ } $H_1(s)$
return $x||u||v$

El punto de partida es $\mathcal{L}^{H_1}_{\text{prg}}$, que se muestra aquí con los detalles de H_1 completados.

$\mathcal{L}_{\text{hyb-1}}$:

QUERY$_{H_1}$():
$x||y :=$ QUERY$_G$()
$u||v := G(y)$
return $x||u||v$

$\diamond$

$\mathcal{L}^G_{\text{prg}}$

QUERY$_G$():
$s \leftarrow \{0,1\}^\lambda$
return $G(s)$

La primera invocación de G se ha factorizado en una subrutina. La librería resultante incluye una instancia de $\mathcal{L}^G_{\text{prg}}$.

$\mathcal{L}_{\text{hyb-2}}$:

QUERY$_{H_1}$():
$x||y :=$ QUERY$_G$()
$u||v := G(y)$
return $x||u||v$

$\diamond$

$\mathcal{L}^G_{\text{prg-\$}}$

QUERY$_G$():
$r \leftarrow \{0,1\}^{2\lambda}$
return r

A partir de la seguridad del PRG G, podemos reemplazar la instancia de $\mathcal{L}^G_{\text{prg}}$ con $\mathcal{L}^G_{\text{prg-\$}}$. La librería resultante es indistinguible de la anterior.

$\mathcal{L}_{\text{hyb-3}}$:

QUERY$_{H_1}$():
$x||y \leftarrow \{0,1\}^{2\lambda}$
$u||v := G(y)$
return $x||u||v$

Se ha integrado una subrutina en línea.

$\mathcal{L}_{\text{hyb-4}}$:

QUERY$_{H_1}$():
$x \leftarrow \{0,1\}^\lambda$
$y \leftarrow \{0,1\}^\lambda$
$u||v := G(y)$
return $x||u||v$

Elegir 2λ bits uniformemente aleatorios y luego dividirlos en dos mitades tiene exactamente el mismo efecto que elegir λ bits uniformemente aleatorios y elegir independientemente λ más.

$\mathcal{L}_{\text{hyb-5}}$:

QUERY$_{H_1}$():
$x \leftarrow \{0,1\}^\lambda$
$u||v :=$ QUERY$_G$()
return $x||u||v$

$\diamond$

$\mathcal{L}^G_{\text{prg}}$

QUERY$_G$():
$s \leftarrow \{0,1\}^\lambda$
return $G(s)$

La aparición restante de G se ha factorizado en una subrutina. Ahora $\mathcal{L}^G_{\text{prg}}$ hace su segunda aparición.

$\mathcal{L}_{\text{hyb-6}}$:

$\underline{\text{QUERY}_{H_1}()}$:
$x \leftarrow \{0,1\}^\lambda$
$u||v := \text{QUERY}_G()$
return $x||u||v$

$\diamond$

$\mathcal{L}^G_{\text{prg-\$}}$

$\underline{\text{QUERY}_G()}$:
$r \leftarrow \{0,1\}^{2\lambda}$
return r

De nuevo, la seguridad del PRG G nos permite reemplazar $\mathcal{L}^G_{\text{prg}}$ con $\mathcal{L}^G_{\text{prg-\$}}$. La librería resultante es indistinguible de la anterior.

$\mathcal{L}_{\text{hyb-7}}$:

$\underline{\text{QUERY}_{H_1}()}$:
$x \leftarrow \{0,1\}^\lambda$
$u||v \leftarrow \{0,1\}^{2\lambda}$
return $x||u||v$

Se ha integrado una subrutina en línea.

$\mathcal{L}^{H_1}_{\text{prg-\$}}$:

$\mathcal{L}^{H_1}_{\text{prg-\$}}$

$\underline{\text{QUERY}_{H_1}()}$:
$r \leftarrow \{0,1\}^{3\lambda}$
return r

Similar a lo anterior, concatenar λ bits uniformes con 2λ bits uniformes independientemente tiene el mismo efecto que muestrear 3λ bits uniformes. El resultado de este cambio es $\mathcal{L}^{H_1}_{\text{prg-\$}}$.

A través de esta secuencia de librerías, hemos mostrado que:

$$\mathcal{L}^{H_1}_{\text{prg}} \equiv \mathcal{L}_{\text{hyb-1}} \approx \mathcal{L}_{\text{hyb-2}} \equiv \mathcal{L}_{\text{hyb-3}} \equiv \mathcal{L}_{\text{hyb-4}} \equiv \mathcal{L}_{\text{hyb-5}} \approx \mathcal{L}_{\text{hyb-6}} \equiv \mathcal{L}_{\text{hyb-7}} \equiv \mathcal{L}^{H_1}_{\text{prg-\$}}.$$

Por lo tanto, H_1 es un PRG seguro.

■

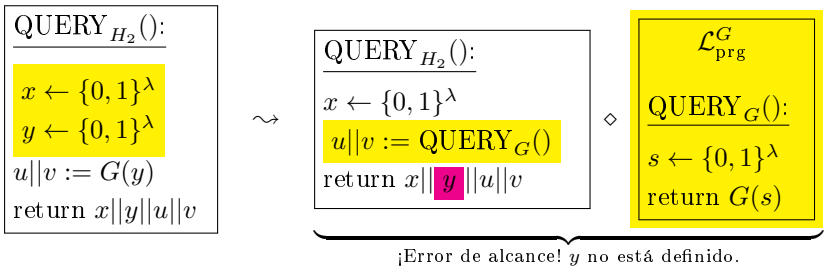
7.4.3. Dónde falla la prueba para H_2

La única diferencia entre H_1 y H_2 es que la variable y se incluye en la salida. ¿Cómo afecta ese pequeño cambio al razonamiento que aplicamos a H_1?

$\underline{H_2(s)}$:
$x \,\|\, y := G(s)$
$u \,\|\, v := G(y)$
return $x||y||u||v$

Anteriormente hemos argumentado que las salidas u y v son indistinguibles de uniforme ya que su entrada y también es indistinguible de aleatoria. Pero no es tan simple: la salida de un PRG es indistinguible de aleatoria si (1) su semilla es uniforme, y (2) ¡la semilla no se usa para nada más! Esta construcción H_2 viola la condición (2) porque incluye la "semilla" y en la salida.

Podemos ver esta idea reflejada en la definición formal de PRG. En $\mathcal{L}_{\text{prg}}$, la semilla s se elige uniformemente, se da como entrada a G, ¡y luego sale del ámbito! Si intentamos reproducir la prueba de seguridad para H_1 con H_2 en su lugar, nos atascaremos cuando intentemos factorizar la segunda llamada a G en términos de $\mathcal{L}_{\text{prg}}$:



Estamos intentando factorizar las dos líneas resaltadas en una librería separada, renombrando y a s en el proceso. Pero s solo puede existir dentro del ámbito privado de la nueva librería, mientras que aún existe una "referencia colgante" y en la librería original.

Hablando de manera más general sobre los PRGs, supongamos que tenemos una llamada a G en alguna parte y queremos argumentar que sus salidas son pseudoaleatorias. Solo podemos expresar esta llamada a G en términos de $\mathcal{L}^G_{\text{prg}}$ si la entrada a G es uniforme y no se usa en ningún otro lugar. Eso no es cierto aquí; no podemos expresar una de las llamadas a G en términos de $\mathcal{L}^G_{\text{prg}}$, por lo que no podemos estar seguros de que las salidas de esa llamada a G parezcan aleatorias.

Estos problemas sutiles no se limitan a los PRGs. Cada prueba de seguridad en este curso incluye pasos en los que factorizamos algunas declaraciones en términos de alguna librería preexistente. ¡No des por sentados estos pasos! Fallarán (a menudo debido a problemas de ámbito) si la construcción no está utilizando el bloque de construcción correctamente. Siempre deben tratar tales pasos de "factorización" como "comprobaciones de cordura" para su prueba.

7.4.4. Un ataque concreto contra H_2

Hasta ahora, solo hemos demostrado que nos atascamos al intentar probar la seguridad de H_2. Pero eso no significa necesariamente que H_2 sea inseguro; podría significar que simplemente no somos lo suficientemente inteligentes para ver una prueba de seguridad diferente. Para demostrar que H_2 es realmente inseguro, debemos demostrar un ataque de distinción exitoso.

Atacar un PRG equivale a encontrar "patrones" en sus salidas. ¿Tiene H_2 un patrón en sus salidas? Sí, en este caso el patrón es que si escribes la salida en la forma $x \parallel y \parallel u \parallel v$, entonces $u \parallel v$ siempre es igual a $G(y)$. Un programa que llama a H_2 puede verificar esta condición, lo cual es poco probable que ocurra para valores verdaderamente uniformes.

Te puedes preguntar, ¿es legal que el programa que llama calcule $G(y)$? Bueno, G es un algoritmo conocido públicamente (¡principio de Kerckhoffs!), e y está ahí mismo como parte de la salida. Nada impide que el programa que llama ejecute G "en su cabeza".[3]

Afirmación 7.4.2. *La construcción H_2 no es un PRG seguro, incluso si G lo es.*

Demostración. Considera el siguiente distinguirdor $\mathcal{A}$:

[3]Compárese con el caso de distinguir $G(s)$ de una cadena de bits escogida de manera uniforme, para un G seguro. El programa que llama a G conoce el algoritmo G pero no tiene la semilla s; solo conoce la salida $G(s)$. En el caso de H_2, el programa que realiza la llamada aprende tanto y como $G(y)$.

$x \parallel y \parallel u \parallel v :=$ QUERY()
return $G(y) \stackrel{?}{=} u \parallel v$

Cuando $\mathcal{A}$ está enlazado a $\mathcal{L}^{H_2}_{\text{prg}}$, las salidas de hecho satisfacen la condición $G(y) = u \parallel v$, por lo que $\mathcal{A}$ devuelve true con probabilidad 1.

Cuando $\mathcal{A}$ está enlazado a $\mathcal{L}^{H_2}_{\text{prg-\$}}$, las salidas son verdaderamente uniformes. Es útil imaginar que x e y se eligen antes que u y v. Tan pronto como se elige y, el valor $G(y)$ se determina de forma única, ya que G es un algoritmo determinista. Entonces $\mathcal{A}$ devolverá true si $u \parallel v$ se elige exactamente para ser igual a este $G(y)$. Dado que u y v se eligen uniformemente, y tienen una longitud total de 2λ bits, este evento ocurre con una probabilidad de $1/2^{2\lambda}$.

La ventaja de $\mathcal{A}$ es la diferencia en estas probabilidades: $1 - 1/2^{2\lambda}$, que no es despreciable. ■

Discusión

En el ataque a H_2, nunca intentamos distinguir la salida de G de uniforme. H_2 es inseguro incluso si G es el mejor PRG del mundo. ¡Es inseguro debido a la forma incorrecta en que usa G!

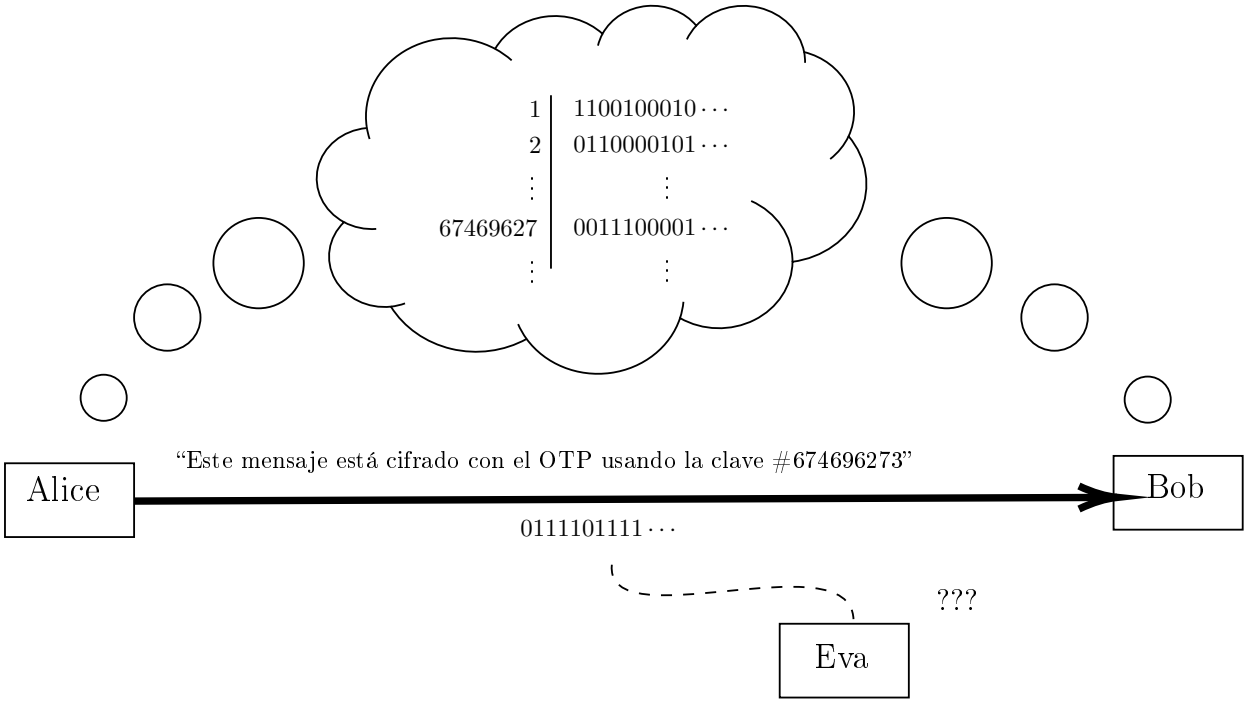
A partir de ahora en este curso, estudiaremos construcciones de nivel superior que se ensamblan a partir de varios bloques de construcción; en este capítulo, PRGs sofisticados construidos a partir de PRGs más simples. "Seguridad" significa: si los bloques de construcción son seguros, entonces la construcción es segura. "Inseguridad" significa: incluso si los bloques de construcción son seguros, la construcción puede ser insegura. Entonces, cuando demuestres inseguridad, ¡no debes atacar directamente los bloques de construcción! Debes asumir que los bloques de construcción son seguros y atacar la forma en que se utilizan los bloques de construcción.

Capítulo 8

Funciones pseudoaleatorias y cifrados en bloque

Imagina que Alice y Bob tuvieran una cantidad infinita de aleatoriedad compartida — no solo una clave corta. Podrían dividirla en trozos de λ bits y usar cada uno como una clave para el one-time pad cada vez que quieran enviar un mensaje cifrado de longitud λ.

Alice podría cifrar diciendo: "Oye Bob, este mensaje está cifrado con el one-time pad usando el trozo #674696273 como clave". Bob podría descifrar buscando la ubicación #674696273 en su copia de la aleatoriedad compartida. Siempre y cuando Alice no repita una clave/trozo, un espía (que no tiene la aleatoriedad compartida) no aprendería nada sobre los mensajes cifrados. Aunque Alice anuncia (públicamente) qué ubicación/trozo se usó como cada clave del one-time pad, esa información no ayuda al atacante a conocer el valor en esa ubicación.



Es absurdo imaginar una cantidad infinita de aleatoriedad compartida. Sin embargo, una cantidad exponencial de algo a menudo es tan buena como una cantidad infinita. Una tabla compartida que contenga "solo" 2^λ claves de un one-time pad sería bastante útil para cifrar tantos mensajes como se pudieran necesitar.

Una función pseudoaleatoria (PRF) es una herramienta que permite a Alice y Bob lograr el efecto de una tabla de aleatoriedad compartida exponencialmente grande en la práctica. En este capítulo exploraremos las PRFs y sus propiedades. En un capítulo posterior, después de introducir nuevas definiciones de seguridad para el cifrado, veremos que las PRFs se pueden usar para cifrar de forma segura *muchos* mensajes con la misma clave, siguiendo la idea principal ilustrada anteriormente.

8.1. Definición

Continuando con nuestro ejemplo, imagina una enorme tabla de datos compartidos almacenada como un array T, de modo que el elemento i-ésimo se referencia como $T[i]$. En lugar de pensar en i como un entero, también podemos pensar en i como una cadena binaria. Si el array tiene 2^{in} elementos, entonces i será una cadena de in bits. Si el array contiene cadenas de longitud out, entonces la notación $T[i]$ es como una función que toma una entrada de $\{0,1\}^{in}$ y da una salida de $\{0,1\}^{out}$.

Una función pseudoaleatoria emula la funcionalidad de un array enorme. Es una función F que toma una entrada de $\{0,1\}^{in}$ y da una salida de $\{0,1\}^{out}$. Sin embargo, F también toma un argumento adicional llamado la **semilla**, que actúa como una especie de clave secreta.

El objetivo de una función pseudoaleatoria es "parecerse" a un array/tabla de búsqueda elegido uniformemente. Se puede acceder a dicho array a través de la subrutina LOOKUP de la siguiente librería:

for $x \in \{0,1\}^{in}$:
  $T[x] \leftarrow \{0,1\}^{out}$

LOOKUP($x \in \{0,1\}^{in}$):
return $T[x]$

Esta librería inicialmente rellena el array T con datos uniformemente aleatorios y luego permite que el programa que la llama acceda a cualquier posición del array.

Una función pseudoaleatoria debería producir un comportamiento indistinguible cuando se usa con una semilla elegida uniformemente. Más formalmente, la siguiente librería debería ser indistinguible de la anterior:

$k \leftarrow \{0,1\}^{\lambda}$

LOOKUP($x \in \{0,1\}^{in}$):
return $F(k,x)$

Observa que la primera librería muestrea $out \cdot 2^{in}$ bits uniformemente al azar (out bits para cada una de las 2^{in} entradas en la tabla), mientras que la segunda librería muestrea solo λ bits (la misma k se usa para todas las invocaciones de F). Aún así, estamos pidiendo que las dos librerías sean indistinguibles.

Esto es básicamente la definición de una PRF, con una salvedad técnica. Queremos permitir situaciones como $in \geq \lambda$, pero en esos casos la primera librería se ejecuta en tiempo exponencial. Generalmente es conveniente construir nuestras definiciones de seguridad con librerías que se ejecutan en tiempo polinómico.[1] Solucionamos esto aprovechando el hecho de que, sin importar cuán grande se suponga que es la tabla T, un programa que la llama en tiempo polinómico solo accederá a una cantidad polinómica de ella. En cierto sentido, es una "exageración" poblar realmente toda la tabla T por adelantado. En cambio, podemos poblar T de forma diferida/bajo demanda. T inicialmente comienza sin inicializar, y sus valores solo se asignan a medida que el programa que llama los solicita. Esto cambia *cuándo* se muestrea cada $T[x]$ (si es que se muestrea), pero no cambia *cómo* se muestrea (es decir, uniforme e independientemente). Esto también cambia T de ser un array típico a ser un array asociativo ("tabla hash" o estructura de datos "diccionario"), ya que solo mapea un subconjunto de $\{0,1\}^{in}$ a valores en $\{0,1\}^{out}$.

Definición 8.1.1. *Sea $F : \{0,1\}^{\lambda} \times \{0,1\}^{in} \to \{0,1\}^{out}$ una función determinista. Decimos que F es una* ***función pseudoaleatoria (PRF)*** *segura si $\mathcal{L}^F_{prf} \approx \mathcal{L}^F_{prf\text{-}\$}$, donde:*

$\mathcal{L}^F_{prf}$

$k \leftarrow \{0,1\}^{\lambda}$

LOOKUP($x \in \{0,1\}^{in}$)*:*
return $F(k,x)$

$\mathcal{L}^F_{prf\text{-}\$}$

$T :=$ *array asoc. vacío*

LOOKUP($x \in \{0,1\}^{in}$)*:*
if $T[x]$ *no está definido:*
  $T[x] \leftarrow \{0,1\}^{out}$
return $T[x]$

[1] Cuando usamos una función pseudoaleatoria como un componente en otras construcciones, las librerías para la seguridad de PRF aparecerán como programas que llaman a otras librerías. La definición de indistinguibilidad requiere que todos los programas que la llaman se ejecuten en tiempo polinómico.

Discusión, Trampas

El nombre "función pseudoaleatoria" proviene de la perspectiva de ver T no como un array (asociativo), sino como una función $T : \{0,1\}^{in} \to \{0,1\}^{out}$. Hay $2^{out \cdot 2^{in}}$ posibles funciones para T (un número increíblemente grande), y $\mathcal{L}_{\text{prf-\$}}$ elige una "función aleatoria" muestreando uniformemente según sea necesario.

Para cada semilla posible k, la función residual $F(k,\cdot)$ también es una función de $\{0,1\}^{in} \to \{0,1\}^{out}$. Hay "solo" 2^{λ} posibles funciones de este tipo (una para cada elección de k), y $\mathcal{L}_{\text{prf}}$ elige una de estas funciones al azar. En ambos casos, las librerías dan al programa que las llama acceso de entrada/salida a la función que se eligió. Puedes pensar en esto en términos de la imagen de la Sección 7.1, pero en lugar de cadenas, los objetos son funciones.

Observa que incluso en el caso de una "función aleatoria" ($\mathcal{L}_{\text{prf-\$}}$), ¡la función T en sí misma sigue siendo **determinista**! Para ser precisos, esta librería elige una función determinista, uniformemente, del conjunto de todas las funciones deterministas posibles. Pero una vez que hace esta elección, el comportamiento de entrada/salida de T es fijo. Si el programa que la llama ejecuta LOOKUP dos veces con la misma x, recibe el mismo resultado. Lo mismo ocurre en $\mathcal{L}_{\text{prf}}$, ya que F es una función determinista y k se fija durante toda la ejecución. Para evitar esta confusión muy natural, quizás sea mejor pensar en términos de "tablas de búsqueda inicializadas aleatoriamente" en lugar de "funciones aleatorias".

8.1.1. Cómo NO construir una PRF

Podemos apreciar los desafíos involucrados en la construcción de una PRF al observar un enfoque natural que no funciona del todo.

Ejemplo 8.1. Supongamos que tenemos un PRG G que duplica la longitud $G : \{0,1\}^{\lambda} \to \{0,1\}^{2\lambda}$ e intentamos usarlo para construir una PRF F de la siguiente manera:

$$\boxed{\begin{array}{l}\underline{F(k,x)\text{:}} \\ \text{return } G(k) \oplus x\end{array}}$$

Podrías notar que todo lo que hemos hecho es renombrar el algoritmo de cifrado de "pseudo-OTP". Anteriormente hemos argumentado que este algoritmo es un método seguro para el cifrado de un solo uso, y que los textos cifrados resultantes son pseudoaleatorios. ¿Es esto suficiente para una PRF segura? No, podemos atacar la seguridad de esta PRF.

Atacar F significa diseñar un distinguidor que se comporte de la manera más diferente posible en presencia de las dos librerías $\mathcal{L}_{\text{prf}}$ y $\mathcal{L}_{\text{prf-\$}}$. Queremos mostrar que F es insegura incluso si G es un excelente PRG. No debemos intentar basar nuestro ataque en distinguir las salidas de G de aleatorias. En cambio, debemos intentar romper la forma inapropiada en que se usa G para construir una PRF.

El distinguidor debe usar la interfaz de las librerías $\mathcal{L}_{\text{prf}}$ y $\mathcal{L}_{\text{prf-\$}}$, es decir, realizar algunas llamadas a la subrutina LOOKUP y generar 0 o 1 según las respuestas que obtenga. La subrutina LOOKUP toma un argumento, por lo que el distinguidor tiene que elegir qué argumentos usar.

Una observación que podemos hacer es que si un distinguidor ve solo un valor de la forma $G(k) \oplus x$, parecerá pseudoaleatorio. Esto es esencialmente lo que mostramos en la Sección 7.3. Por lo tanto, deberíamos construir un distinguidor que realice más de una llamada a LOOKUP.

Si hacemos dos llamadas a LOOKUP, digamos, con las entradas x_1 y x_2, las respuestas de $\mathcal{L}^F_{\text{prf}}$ serán $G(k) \oplus x_1$ y $G(k) \oplus x_2$. Para ser una PRF segura, estas respuestas deben parecer independientes y uniformes. ¿Lo son? En realidad, tienen un patrón que el distinguidor puede notar: su XOR es siempre $x_1 \oplus x_2$, un valor que ya conoce el distinguidor.

Podemos condensar todas nuestras observaciones en el siguiente distinguidor:

$\mathcal{A}$

elegir $x_1, x_2 \in \{0,1\}^{2\lambda}$ arbitrarias tal que $x_1 \neq x_2$
$z_1 := \text{LOOKUP}(x_1)$
$z_2 := \text{LOOKUP}(x_2)$
return $z_1 \oplus z_2 \stackrel{?}{=} x_1 \oplus x_2$

Calculemos su *ventaja* al distinguir $\mathcal{L}^F_{\text{prf}}$ de $\mathcal{L}^F_{\text{prf-\$}}$, considerando el comportamiento de $\mathcal{A}$ cuando se vincula a estas dos librerías:

$\mathcal{A}$

elegir $x_1 \neq x_2 \in \{0,1\}^{2\lambda}$
$z_1 := \text{LOOKUP}(x_1)$
$z_2 := \text{LOOKUP}(x_2)$
return $z_1 \oplus z_2 \stackrel{?}{=} x_1 \oplus x_2$

$\diamond$

$\mathcal{L}^F_{\text{prf}}$

$k \leftarrow \{0,1\}^{\lambda}$

LOOKUP(x):
return $G(k) \oplus x$ \\ $F(k, x)$

Cuando $\mathcal{A}$ se vincula a $\mathcal{L}^F_{\text{prf}}$, la librería elegirá una clave k. Entonces z_1 se establece en $G(k) \oplus x_1$ y z_2 se establece en $G(k) \oplus x_2$. Así que $z_1 \oplus z_2$ siempre es igual a $x_1 \oplus x_2$, y $\mathcal{A}$ siempre produce 1. Es decir:

$$\Pr[\mathcal{A} \diamond \mathcal{L}^F_{\text{prf}} \Rightarrow 1] = 1.$$

$\mathcal{A}$

elegir $x_1 \neq x_2 \in \{0,1\}^{2\lambda}$
$z_1 := \text{LOOKUP}(x_1)$
$z_2 := \text{LOOKUP}(x_2)$
return $z_1 \oplus z_2 \stackrel{?}{=} x_1 \oplus x_2$

$\diamond$

$\mathcal{L}^F_{\text{prf-\$}}$

$T :=$ array asoc. vacío

LOOKUP(x:
if $T[x]$ no está definido:
  $T[x] \leftarrow \{0,1\}^{2\lambda}$
return $T[x]$

Cuando $\mathcal{A}$ se vincula a $\mathcal{L}^F_{\text{prf-\$}}$, las respuestas de las dos llamadas a LOOKUP se elegirán de forma uniforme e independiente porque se llama a LOOKUP con entradas distintas. Consideremos el momento en que está a punto de ocurrir la segunda llamada a LOOKUP. En ese momento, x_1, x_2 y z_1 ya se han determinado, mientras que z_2 está a punto de ser elegido uniformemente por la librería. Usando las propiedades de XOR, vemos que $\mathcal{A}$ producirá 1 si y solo si z_2 se elige para que sea exactamente el valor $x_1 \oplus x_2 \oplus z_1$. Esto ocurre solo con una probabilidad de $1/2^{\lambda}$. Es decir:

$$\Pr[\mathcal{A} \diamond \mathcal{L}^F_{\text{prf-\$}} \Rightarrow 1] = 1/2^{\lambda}.$$

La ventaja de $\mathcal{A}$ es, por lo tanto, $1 - 1/2^{\lambda}$, que ciertamente no es despreciable, ya que ni siquiera se acerca a 0. Esto muestra que F no es una PRF segura.

En un nivel más filosófico, queríamos identificar exactamente cómo se está utilizando G de una manera inapropiada. Las librerías de seguridad PRG garantizan la seguridad cuando la semilla de G se elige de nuevo para cada llamada a G. Esta construcción de F viola esa regla y permite que se use la misma semilla dos veces en diferentes llamadas a G, donde se supone que los resultados parecen independientes.

Este ejemplo muestra el desafío de construir una PRF. Aunque sabemos cómo hacer que cualquier salida individual sea pseudoaleatoria, es difícil hacer que todas las salidas parezcan colectivamente independientes, cuando en realidad se derivan de una sola semilla corta.

8.2. PRFs vs PRGs

En esta sección mostramos que se puede usar un PRG para construir una PRF. La construcción de un PRG a partir de una PRF es práctica, y es una de las formas más comunes de obtener un PRG en la práctica. La construcción de una PRF a partir de un PRG es de interés más teórico y no refleja cómo se diseñan las PRFs en la práctica.

8.2.1. Construyendo un PRG a partir de una PRF

Como se prometió, se puede usar una PRF para construir un PRG. La construcción es bastante natural. Por simplicidad, supongamos que tenemos una PRF $F : \{0,1\}^{\lambda} \times \{0,1\}^{\lambda} \to \{0,1\}^{\lambda}$ (es decir, $in = out = \lambda$). Podemos construir un PRG que duplica la longitud de la siguiente manera:

PRG contador:

$G(s)$:
$x := F(s, 00\cdots00)$
$y := F(s, 00\cdots01)$
return $x||y$

No hay nada particularmente especial sobre las entradas $0\cdots00$ y $0\cdots01$ para F. Lo único que importa es que sean distintas. La construcción se puede extender para dar fácilmente más de 2 bloques de salida, tratando la entrada a F como un simple contador (de ahí el nombre de esta construcción).

La garantía de una PRF es que cuando su semilla se elige uniformemente y se invoca con entradas distintas, sus salidas parecen uniformemente independientes. En particular, su salida en las entradas $0\cdots00$ y $0\cdots01$ son indistinguibles de uniformes. Por lo tanto, concatenarlas da como resultado una cadena que es indistinguible de una cadena uniforme de 2λ bits.

Eso es realmente todo lo que hay en la seguridad de esta construcción, pero desafortunadamente hay un pequeño problema técnico que hace que la prueba de seguridad sea más complicada de lo que se podría suponer. Tendremos que introducir una nueva técnica de demostración para lidiar con ello.

Afirmación 8.2.1. *Si F es una PRF segura, entonces la construcción del PRG de contador G anterior es un PRG seguro.*

Demostración. Para probar que G es un PRG seguro, debemos probar que las siguientes librerías son indistinguibles:

$\mathcal{L}^G_{\text{prg}}$

QUERY():
$s \leftarrow \{0,1\}^\lambda$
$x := F(s, 00\cdots00)$
$y := F(s, 00\cdots01)$ \\$G(s)$
return $x||y$

$\mathcal{L}^G_{\text{prg-\$}}$

QUERY():
$r \leftarrow \{0,1\}^{2\lambda}$
return r

Durante la prueba, se nos permite usar el hecho de que F es una PRF segura. Es decir, podemos usar el hecho de que las siguientes dos librerías son indistinguibles:

$\mathcal{L}^F_{\text{prf}}$

$k \leftarrow \{0,1\}^\lambda$

LOOKUP($x \in \{0,1\}^{in}$):
return $F(k,x)$

$\mathcal{L}^F_{\text{prf-\$}}$

$T :=$ array asoc. vacío

LOOKUP($x \in \{0,1\}^{in}$):
if $T[x]$ no está definido:
$T[x] \leftarrow \{0,1\}^{out}$
return $T[x]$

El inconveniente en la prueba surge de una discrepancia entre la variable s en $\mathcal{L}^G_{\text{prg}}$ y la variable k en $\mathcal{L}^G_{\text{prf-\$}}$. En $\mathcal{L}^G_{\text{prg}}$, s es local a la subrutina QUERY. A lo largo de una ejecución, s tomará muchos valores. Dado que s se usa como la semilla de la PRF, debemos escribir las llamadas a F en términos de la subrutina LOOKUP de $\mathcal{L}^F_{\text{prf}}$. Pero en $\mathcal{L}^F_{\text{prf}}$ la semilla de la PRF se fija para toda la ejecución. En otras palabras, solo podemos usar $\mathcal{L}^F_{\text{prf}}$ para tratar con una sola semilla de PRF a la vez, pero $\mathcal{L}^G_{\text{prg}}$ trata con muchas semillas de PRG a la vez.

Para abordar esto, tendremos que aplicar la seguridad de F (es decir, reemplazar $\mathcal{L}^F_{\text{prf}}$ con $\mathcal{L}^F_{\text{prf-\$}}$ muchas veces durante la prueba; de hecho, una vez por cada llamada a QUERY realizada por el programa que las llama. Las pruebas de seguridad anteriores tenían un número fijo de pasos intermedios (por ejemplo, la prueba de la Afirmación 7.4.1 usó 7 librerías intermedias para mostrar $\mathcal{L}^G_{\text{prg}} \approx \mathcal{L}_{\text{hyb-1}} \approx \cdots \approx \mathcal{L}_{\text{hyb-7}} \approx \mathcal{L}_{\text{prg-\$}}$). Esta prueba tendrá un **número variable de librerías intermedias que depende del programa que las llama**. Específicamente, probaremos

$$\mathcal{L}^G_{\text{prg}} \approx \mathcal{L}_{\text{hyb-1}} \cdots \approx \mathcal{L}_{\text{hyb-}q} \approx \mathcal{L}^G_{\text{prg-\$}}$$

donde q es el número de veces que el programa llama a QUERY.

No te abrumes por todas estas librerías. Todos siguen un patrón simple. De hecho, la i-ésima librería se ve así:

$\mathcal{L}_{\text{hyb-}i}$

count:=0

QUERY():
count:=count+1
if count$\leq i$:
  $r \leftarrow \{0,1\}^{2\lambda}$
  return r
else:
  $s \leftarrow \{0,1\}^{\lambda}$
  $x := F(s, 00\cdots00)$
  $y := F(s, 00\cdots01)$
  return $x||y$

En otras palabras, las librerías intermedias difieren todas en el valor i que se inserta en el código anterior.

Primero, observa lo que sucede para elecciones extremas de i:

- En $\mathcal{L}_{\text{hyb-0}}$, la rama *if* nunca se toma (count≤ 0 nunca es verdadero porque count≥ 1). Esta librería se comporta exactamente como $\mathcal{L}^G_{\text{prg}}$ al dar salidas de PRG en cada llamada a QUERY.
- Si q es el número total de veces que el programa que llama llama a QUERY, entonces en $\mathcal{L}_{\text{hyb-}q}$, la rama *if* siempre se toma ($count \leq q$ siempre es verdadero). Esta librería se comporta exactamente como $\mathcal{L}^G_{\text{prg-\$}}$ al dar una salida verdaderamente uniforme en cada llamada a QUERY.

En general, $\mathcal{L}_{\text{hyb-}i}$ responderá a las primeras i llamadas a QUERY dando una salida verdaderamente aleatoria. Responderá a todas las llamadas posteriores dando salidas de nuestra construcción de PRG.

Hemos argumentado que $\mathcal{L}^G_{\text{prg}} = \mathcal{L}_{\text{hyb-0}}$ y $\mathcal{L}^G_{\text{prg-\$}} = \mathcal{L}_{\text{hyb-}q}$. Para completar la prueba, debemos mostrar que $\mathcal{L}_{\text{hyb-}(i-1)} \approx \mathcal{L}_{\text{hyb-}i}$ para todo i. La razón principal para pasar por toda esta molestia de definir tantas librerías intermedias es que $\mathcal{L}_{\text{hyb-}(i-1)}$ y $\mathcal{L}_{\text{hyb-}i}$ son completamente idénticas excepto en cómo responden a la i-ésima llamada a `QUERY`. Esta diferencia implica una sola llamada al PRG (y por lo tanto una sola semilla de PRF), lo que nos permite aplicar la seguridad de la PRF.

Con más detalle, sea i arbitrario, y considera la siguiente secuencia de pasos comenzando con $\mathcal{L}_{\text{hyb-}(i-1)}$:

count:=0

QUERY():
count:=count+1
if count< i :
  $r \leftarrow \{0,1\}^{2\lambda}$
  return r
elseif count= i :
  $s^* \leftarrow \{0,1\}^{\lambda}$
  $x := F(s^*, 00\cdots00)$
  $y := F(s^*, 00\cdots01)$
  return $x||y$
else:
  $s \leftarrow \{0,1\}^{\lambda}$
  $x := F(s, 00\cdots00)$
  $y := F(s, 00\cdots01)$
  return $x||y$

Hemos tomado $\mathcal{L}_{\text{hyb-}(i-1)}$ y simplemente hemos expandido la rama *else* ($count \leq i$) en dos subcasos ($count = i$ y $count < i$). Sin embargo, ambos casos conducen al mismo bloque de código (aparte de un cambio en el nombre de una variable local), por lo que el cambio no tiene ningún efecto en el programa que llama las librerías.

count:=0

QUERY():
count:=count+1
if count< i :
  $r \leftarrow \{0,1\}^{2\lambda}$
  return r
elseif count= i :
  $x :=$ LOOKUP$(00\cdots00)$
  $y :=$ LOOKUP$(00\cdots01)$
  return $x||y$
else:
  $s \leftarrow \{0,1\}^{\lambda}$
  $x := F(s, 00\cdots00)$
  $y := F(s, 00\cdots01)$
  return $x||y$

$\diamond$

$\mathcal{L}^F_{\text{prf}}$
$k \leftarrow \{0,1\}^{\lambda}$

LOOKUP(x):
return $F(k,x)$

Hemos factorizado las llamadas a F que usan la semilla s^* (correspondiente al caso $count = i$) en términos de $\mathcal{L}_{\text{prf}}$. Este cambio no tiene ningún efecto en el programa que llama.

count:=0

QUERY():
count:=count+1
if count< i:
  $r \leftarrow \{0,1\}^{2\lambda}$
  return r
elseif count= i:
  $x :=$ LOOKUP$(00\cdots00)$
  $y :=$ LOOKUP$(00\cdots01)$
  return $x||y$
else:
  $s \leftarrow \{0,1\}^{\lambda}$
  $x := F(s, 00\cdots00)$
  $y := F(s, 00\cdots01)$
  return $x||y$

$\diamond$

$\mathcal{L}^F_{\text{prf-\$}}$

$T :=$ array asoc. vacío

LOOKUP(x):
if $T[x]$ no está definido:
  $T[x] \leftarrow \{0,1\}^{out}$
return $T[x]$

Del hecho de que F es una PRF segura, podemos reemplazar $\mathcal{L}^F_{\text{prf}}$ con $\mathcal{L}^F_{\text{prf-\$}}$, y el cambio general es indistinguible.

count:=0

QUERY():
count:=count+1
if count< i:
  $r \leftarrow \{0,1\}^{2\lambda}$
  return r
elseif count= i:
  $x :=$ LOOKUP$(00\cdots00)$
  $y :=$ LOOKUP$(00\cdots01)$
  return $x||y$
else:
  $s \leftarrow \{0,1\}^{\lambda}$
  $x := F(s, 00\cdots00)$
  $y := F(s, 00\cdots01)$
  return $x||y$

$\diamond$

LOOKUP(x):
$r \leftarrow \{0,1\}^{\lambda}$
return r

Dado que count $= i$ ocurre solo una vez, solo se realizarán dos llamadas a LOOKUP durante toda la vida útil de la librería, y están en entradas distintas. Por lo tanto, la rama *if* en LOOKUP siempre se tomará, y T nunca es necesaria (solo se necesita para "recordar" valores y dar la misma respuesta cuando la misma x se usa dos veces como argumento para LOOKUP). Simplificar la librería, por lo tanto, no tiene ningún efecto en el programa que llama.

count:=0

QUERY():
count:=count+1
if count< i :
  $r \leftarrow \{0,1\}^{2\lambda}$
  return r
elseif count= i :
  $x \leftarrow \{0,1\}^{\lambda}$
  $y \leftarrow \{0,1\}^{\lambda}$
  return $x||y$
else:
  $s \leftarrow \{0,1\}^{\lambda}$
  $x := F(s, 00\cdots00)$
  $y := F(s, 00\cdots01)$
  return $x||y$

Incluir la subrutina en el código no tiene ningún efecto en el programa vinculado. La librería resultante responde con una salida uniformemente aleatoria a las primeras i llamadas a QUERY y responde con salidas de nuestro PRG G a las demás. Por lo tanto, esta librería tiene un comportamiento idéntico a $\mathcal{L}_{\text{hyb-}i}$.

Mostramos que $\mathcal{L}_{\text{hyb-}(i-1)} \approx \mathcal{L}_{\text{hyb-}i}$, y por lo tanto:

$$\mathcal{L}^G_{\text{prg}} = \mathcal{L}_{\text{hyb-0}} \approx \mathcal{L}_{\text{hyb-1}} \approx \cdots \approx \mathcal{L}_{\text{hyb-}q} = \mathcal{L}^G_{\text{prg-\$}}$$

Esto muestra que $\mathcal{L}^G_{\text{prg}} \approx \mathcal{L}^G_{\text{prg-\$}}$, por lo que G es un PRG seguro.

■

8.3. Cifrados en bloque (permutaciones pseudoaleatoria)

Después de fijar la semilla, una PRF calcula una función de $\{0,1\}^{in}$ a $\{0,1\}^{out}$. Consideremos el caso donde $in = out$. Algunas funciones de $\{0,1\}^{in}$ a $\{0,1\}^{out}$ son invertibles, lo que lleva a la pregunta de si una PRF podría realizar tal función y ser invertible (con conocimiento de la semilla). En otras palabras, ¿qué pasaría si fuera posible determinar x cuando se dan k y $F(k,x)$? Si bien esta sería una propiedad conveniente, no está garantizada por la definición de seguridad de la PRF, incluso en el caso de $in = out$. Es poco probable que una función de $\{0,1\}^{in}$ a $\{0,1\}^{out}$ elegida al azar tenga una inversa; por lo tanto, es poco probable que una PRF instanciada con una clave aleatoria tenga una inversa.

Una **permutación pseudoaleatoria (PRP)**, también llamada **cifrado en bloque**, es esencialmente una PRF que garantiza ser invertible para cada elección de semilla. Usamos ambos términos (PRP y cifrado en bloque) indistintamente. El término "permutación" se refiere al hecho de que, para cada k, la función $F(k,\cdot)$ debe ser una permutación de $\{0,1\}^n$. En lugar de requerir que una PRP sea indistinguible de una función elegida al azar, requerimos que sea indistinguible de una función *invertible* elegida al azar.[2] Esto significa que debemos modificar una de las librerías de la definición de PRF. En lugar de completar el array asociativo T

[2]Como veremos más adelante, la distinción entre una función elegida aleatoriamente y una función invertible elegida aleatoriamente no es tan significativa como podría parecer.

con valores uniformemente aleatorios, elige valores uniformemente aleatorios pero **distintos**. Siempre que T proporcione salidas distintas en entradas distintas, es consistente con alguna función invertible. Las librerías garantizan la distinción muestreando solo valores que no se han asignado previamente. Pensando en un array asociativo T como un almacén de clave-valor, usamos la notación $T.\texttt{values}$ para denotar el conjunto de valores almacenados en T.

Definición 8.3.1. *Sea $F : \{0,1\}^\lambda \times \{0,1\}^{blen} \to \{0,1\}^{blen}$ una función determinista. Nos referimos a blen como la longitud de bloque de F y a cualquier elemento de $\{0,1\}^{blen}$ como un bloque.*

Llamamos a F una permutación pseudoaleatoria (PRP) segura (cifrado en bloque) si se cumplen las siguientes dos condiciones:

1. ***(Invertible dado k)** Existe una función $F^{-1} : \{0,1\}^\lambda \times \{0,1\}^{blen} \to \{0,1\}^{blen}$ que satisface*

$$F^{-1}(k, F(k,x)) = x,$$

   para toda $k \in \{0,1\}^\lambda$ y toda $x \in \{0,1\}^{blen}$.

2. ***(Seguridad)** $\mathcal{L}^F_{prp} \approx \mathcal{L}^F_{prp\text{-}\$}$, donde:*

$\mathcal{L}^F_{prp}$

$k \leftarrow \{0,1\}^\lambda$

$\underline{LOOKUP(x \in \{0,1\}^{blen}):}$
return $F(k,x)$

$\mathcal{L}^F_{prp\text{-}\$}$

$T :=$ *array asoc. vacío*

$\underline{LOOKUP(x \in \{0,1\}^{blen}):}$
if $T[x]$ *no está definido:*
  $T[x] \leftarrow \{0,1\}^{blen} \setminus T.\texttt{values}$
return $T[x]$

$T.\texttt{values}$ se refiere al conjunto $\{v \mid \exists x : T[x] = v\}$.

Los cambios con respecto a la definición de PRF están resaltados en amarillo. En particular, las librerías $\mathcal{L}_{\text{prp}}$ y $\mathcal{L}_{\text{prf}}$ son idénticas.

Discusión, Trampas

En la definición, tanto las funciones F como F^{-1} toman la semilla k como entrada. Por lo tanto, solo alguien con k puede invertir el cifrado en bloque. Recordemos la definición de una PRF: sin la semilla k, es difícil calcular $F(k,x)$. ¡Un cifrado en bloque tiene una dirección hacia adelante y una inversa, y calcular cualquiera de ellas es difícil sin k!

8.4. Relacionando PRFs y cifrados en bloque

En esta sección discutimos cómo obtener PRFs a partir de PRPs/cifrados en bloque, y viceversa.

8.4.1. Lema de conmutación (¡Las PRPs también son PRFs!)

Imagina que puedes consultar una PRP en entradas elegidas (como en la librería $\mathcal{L}^F_{\text{prp}}$), y suponga que la longitud de bloque de la PRP es $blen = \lambda$. Solo podrías consultar esa PRP en una *fracción insignificante*

de su dominio de entrada exponencialmente grande. Parece poco probable que incluso puedas decir que era una PRP (es decir, una función invertible) en lugar de una PRF (una función sin restricciones).

Esta idea se puede formalizar de la siguiente manera.

Lema 8.4.1 (Conmutación de PRP). *Sean $\mathcal{L}_{prf\text{-}\$}$ y $\mathcal{L}_{prp\text{-}\$}$ definidas como en las Definiciones 8.1.1 y 8.3.1, con parámetros $in = out = blen = \lambda$ (para que las interfaces coincidan). Entonces $\mathcal{L}_{prf\text{-}\$} \approx \mathcal{L}_{prp\text{-}\$}$.*

Demostración. Recuerde el lema de muestreo con reemplazo, Lema 6.4.3, que mostró que las siguientes librerías son indistinguibles:

$\mathcal{L}_{\text{samp-0}}$

SAMP():
$r \leftarrow \{0,1\}^\lambda$
return r

$\mathcal{L}_{\text{samp-1}}$

$R := \emptyset$

SAMP():
$r \leftarrow \{0,1\}^\lambda \setminus R$
$R := R \cup \{r\}$
return r

$\mathcal{L}_{\text{samp-0}}$ muestrea valores con reemplazo, y $\mathcal{L}_{\text{samp-1}}$ muestrea valores sin reemplazo. Ahora considera la siguiente librería $\mathcal{L}^*$:

$\mathcal{L}^*$

$T :=$ array asoc. vacío

LOOKUP($x \in \{0,1\}^\lambda$):
if $T[x]$ no está definido:
    $T[x] \leftarrow$ SAMP()
return $T[x]$

Cuando enlazamos $\mathcal{L}^*$ con $\mathcal{L}_{\text{samp-0}}$, obtenemos $\mathcal{L}^F_{\text{prf-\$}}$ ya que los valores en $T[x]$ se muestrean uniformemente. Cuando enlazamos $\mathcal{L}^*$ con $\mathcal{L}_{\text{samp-1}}$, obtenemos $\mathcal{L}_{\text{prp-\$}}$ ya que los valores en $T[x]$ se muestrean uniformemente sujetos a no tener repeticiones (considere que R desempeña el papel de $T.\texttt{values}$ en $\mathcal{L}^F_{\text{prp-\$}}$). Entonces, del Lema 6.4.3, tenemos:

$$\mathcal{L}_{\text{prf-\$}} \equiv \mathcal{L}^* \diamond \mathcal{L}_{\text{samp-0}} \approx \mathcal{L}^* \diamond \mathcal{L}_{\text{samp-1}} \equiv \mathcal{L}^F_{\text{prp-\$}},$$

lo que completa la prueba. ■

Usando el lema de conmutación, podemos concluir que toda PRP (con $blen = \lambda$) también es una PRF:

Corolario 8.4.2. *Sea $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$ una PRP segura (con $blen = \lambda$). Entonces F también es una PRF segura.*

Demostración. Como hemos observado anteriormente, $\mathcal{L}^F_{\text{prf}}$ y $\mathcal{L}^F_{\text{prp}}$ son literalmente la misma librería. Dado que F es una PRP segura, $\mathcal{L}^F_{\text{prp}} \approx \mathcal{L}^F_{\text{prp-\$}}$. Finalmente, por el lema de conmutación, $\mathcal{L}^F_{\text{prp-\$}} \approx \mathcal{L}^F_{\text{prf-\$}}$. Juntando todo:

$$\mathcal{L}^F_{\text{prf}} \equiv \mathcal{L}^F_{\text{prp}} \approx \mathcal{L}^F_{\text{prp-\$}} \approx \mathcal{L}^F_{\text{prf-\$}}$$

por lo tanto, F es una PRF segura. ■

Ten en cuenta que el lema de conmutación se aplica solo cuando la longitud del bloque es suficientemente grande (al menos λ bits de longitud). Esto proviene del hecho de que $\mathcal{L}_{\text{samp-0}}$ y $\mathcal{L}_{\text{samp-1}}$ en la prueba son indistinguibles solo cuando se muestrean cadenas largas (de longitud λ) (consulte la prueba del Lema 6.4.3 para recordar por qué).

8.5. Construyendo una PRP a partir de una PRF: La construcción de Feistel

¿Cómo se puede construir un cifrado en bloque invertible a partir de una PRF que no es necesariamente invertible?

En esta sección, mostramos una técnica simple llamada la *construcción de Feistel* (llamada así por el criptógrafo de IBM Horst Feistel).

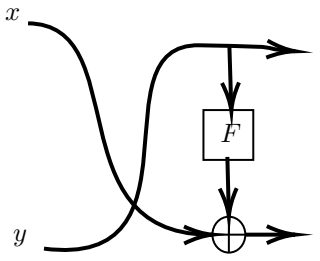
La idea principal en la construcción de Feistel es convertir una función $F : \{0,1\}^n \to \{0,1\}^n$ que no es necesariamente invertible en una función invertible $F^* : \{0,1\}^{2n} \to \{0,1\}^{2n}$. La función F^* se llama la **ronda de Feistel** con la función de ronda F y se define de la siguiente manera:

Ronda de Feistel:

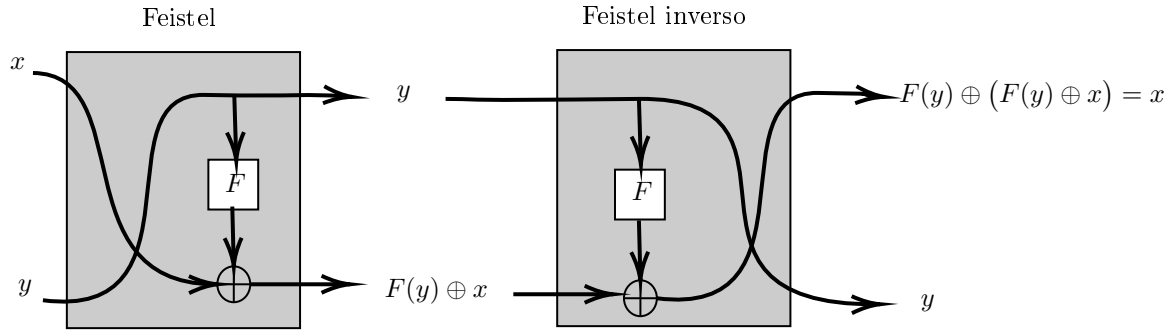
$F^*(x||y)$:

$\backslash\backslash$ x, y tienen n bits cada una

return $y||(F(y) \oplus x)$



No importa lo que sea F, su ronda de Feistel F^* es invertible. No solo eso, sino que su inversa es una especie de "imagen especular" de F^*:



Observe cómo tanto la ronda de Feistel directa como la inversa usan F en la dirección directa.

Ejemplo 8.2. Veamos qué sucede en la construcción de Feistel con una función de ronda trivial. Considere la función constante $F(y) = 0^n$ que es la función "menos invertible" imaginable. La construcción de Feistel da como resultado:

$$\begin{aligned} F^*(x||y) &= y||(F(y) \oplus x) \\ &= y||(0^n \oplus x) \\ &= y||x \end{aligned}$$

El resultado es una función que simplemente intercambia el orden de sus mitades - claramente invertible.

Ejemplo 8.3. Intentemos con otra función de ronda simple, esta vez la función identidad $F(y) = y$. La construcción de Feistel da como resultado:

$$\begin{aligned} F^*(x||y) &= y||(F(y) \oplus x) \\ &= y||(y \oplus x) \end{aligned}$$

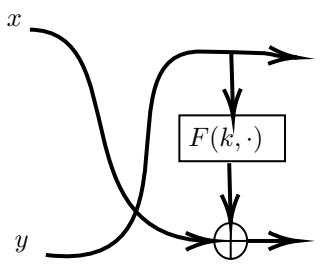
Esta función es invertible porque dados y e $y \oplus x$ podemos resolver para x como $y \oplus (y \oplus x)$. Puede verificar que esto es lo que sucede cuando se introduce F en la construcción de Feistel inversa.

También podemos considerar el uso de una función de ronda F que tenga una clave/semilla. El resultado será un F^* que también toma una semilla. Para cada semilla k, $F^*(k, \cdot)$ tendrá una inversa (que se ve como su imagen especular).

Feistel con clave:

$F^*(k,\ x||y)$:

return $y||(F(k, y) \oplus x$



Ahora suponga que F es una PRF segura y la usamos como una función de ronda de Feistel, para obtener una función con clave F^*. Dado que $F^*(k, \cdot)$ es invertible para cada k, y dado que F^* usa una PRF segura

de alguna manera, podrías verte tentado a afirmar que F^* es una PRP segura. ¡Desafortunadamente, no lo es! La salida de F^* contiene la mitad de su entrada, lo que hace que sea bastante trivial romper la seguridad de PRP de F^*.

Podemos evitar este ataque trivial realizando varias rondas de Feistel sucesivamente, lo que resulta en una construcción llamada *cifrado de Feistel*. En cada ronda, incluso podemos usar una clave diferente para la función de ronda. Si usamos k_1 en la primera ronda, k_2 en la segunda ronda, y así sucesivamente, entonces $k_1, k_2, ...$ se llama el **programa de claves** del cifrado de Feistel. La definición formal de un cifrado de Feistel de r rondas se da a continuación:

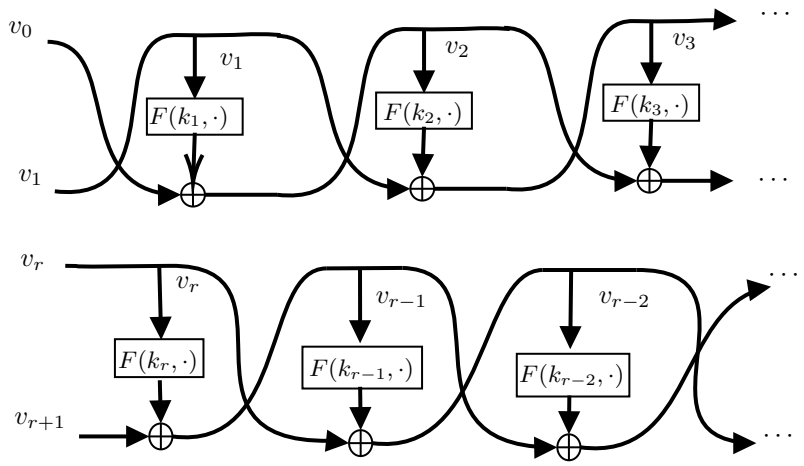
Cifrado de Feistel:

$\mathbb{F}_r((k_1, \ldots, k_r),\ v_0||v_1)$:

for $i = 1$ to r:
$v_{i+1} := F(k_i, v_i) \oplus v_{i-1}$
return $v_r||v_{r+1}$

$\mathbb{F}_r^{-1}((k_1, \ldots, k_r),\ v_r||v_{r+1})$:

for $i = r$ downto 1:
$v_{i-1} := F(k_i, v_i) \oplus v_{i+1}$
return $v_0||v_1$



Debido a que cada ronda es invertible (dada la clave de ronda apropiada), el cifrado de Feistel general también es invertible. Observa que la inversa del cifrado de Feistel usa rondas de Feistel inversas e invierte el orden del programa de claves.

Sorprendentemente, un cifrado de Feistel de 3 rondas en realidad puede ser seguro, aunque un cifrado de Feistel de 2 rondas nunca es seguro. Más precisamente: cuando F es una PRF segura con $in = out = \lambda$, entonces usar F como la función de ronda de un cifrado de Feistel de 3 rondas da como resultado una PRP segura. El cifrado de Feistel tiene una longitud de bloque de 2λ y tiene una clave de longitud 3λ (3 veces más larga que la clave para F). Implícitamente, esto significa que las tres claves de ronda se eligen de forma independiente.

Teorema 8.5.1 (Luby-Rackoff). *Si $F : \{0,1\}^\lambda \times \{0,1\}^\lambda \to \{0,1\}^\lambda$ es una PRF segura, entonces el cifrado de Feistel de 3 rondas $\mathbb{F}_3$ es una PRP segura.*

Desafortunadamente, la prueba de este teorema está fuera del alcance de este curso.

8.6. PRFs y cifrados en bloque en la práctica

Los cifrados en bloque son una de las piedras angulares de la criptografía en la práctica actual. Hemos mostrado cómo (al menos en principio) los cifrados en bloque se pueden construir a partir de primitivas más simples: PRGs y PRFs. Sin embargo, en la práctica utilizamos cifrados en bloque que se diseñan "desde cero" y luego utilizamos estos cifrados en bloque para construir PRGs y PRFs más simples cuando los necesitamos.

Actualmente **no tenemos ninguna prueba** de que exista alguna PRP segura. Como discutimos en la Sección 7.2, tal prueba resolvería el famoso problema P vs NP. Sin tales pruebas, ¿cuál es nuestra base para la confianza en la seguridad de los cifrados en bloque que se utilizan hoy en día? El proceso que condujo al cifrado en bloque *Advanced Encryption Standard* (AES) demuestra los mejores esfuerzos de la comunidad criptográfica para infundir tal confianza.

El National Institute of Standards & Technology (NIST) patrocinó un concurso para diseñar un cifrado en bloque que reemplazara el estándar DES de la década de 1970. Muchos equipos de criptógrafos presentaron sus diseños de cifrado en bloque, todos los cuales fueron sometidos a años de intenso escrutinio público por parte de la comunidad de investigación criptográfica. Los diseños se evaluaron en función de su rendimiento y resistencia a los ataques contra la definición de seguridad PRP (y otros ataques). Algunos diseños ofrecieron pruebas de que resisten ciertas clases de ataques y pruebas que justifican ciertas elecciones en la construcción del cifrado en bloque a partir de componentes más simples.

El cifrado Rijndael, diseñado por Vincent Rijmen y Joan Daemen, fue seleccionado como ganador y se convirtió en el estándar AES en 2001. Puede que no haya otro algoritmo criptográfico que haya sido el foco de más escrutinio e intentos de ataque. Hasta ahora no se conocen debilidades significativas en AES.

El cifrado en bloque AES tiene una longitud de bloque de 128 bits y ofrece 3 variantes diferentes con claves de 128 bits, 192 bits y 256 bits. Como resultado de su estandarización, AES está disponible en librerías criptográficas para cualquier lenguaje de programación. Incluso se implementa como instrucciones de hardware en la mayoría de los procesadores modernos, lo que permite millones de evaluaciones de AES por segundo. Como hemos visto, una vez que se tiene acceso a un buen cifrado en bloque, también se puede usar directamente como una PRF segura (Corolario 8.4.2), y se puede usar para construir un PRG simple (PRG contador de la Sección 8.2.1). Aunque AES en sí mismo no es una PRP demostrablemente segura, estas construcciones de PRFs y PRGs basadas en AES son seguras. O, más precisamente, la seguridad PRF y la seguridad PRG de estas construcciones están garantizadas para ser tan buenas como la seguridad PRP de AES.

Capítulo 9

Seguridad frente a ataques de texto plano elegido

Nuestras definiciones de seguridad anteriores para el cifrado capturan el caso en el que se usa una clave para cifrar solo un texto plano. Claramente, sería más útil tener un esquema de cifrado que permita cifrar muchos textos planos con la misma clave.

Afortunadamente, hemos organizado las cosas de tal manera que obtenemos la definición de seguridad "correcta" cuando modificamos la definición anterior de una manera natural. Simplemente dejamos que las librerías elijan una clave secreta de una vez por todas, que se usa para cifrar todos los textos planos. Más formalmente:

Definición 9.0.1 (Seguridad CPA)**.** *Sea Σ un esquema de cifrado. Decimos que Σ tiene seguridad contra ataques de texto plano elegido (seguridad CPA) si $\mathcal{L}^{\Sigma}_{cpa\text{-}0} \approx \mathcal{L}^{\Sigma}_{cpa\text{-}1}$ donde:*

$\mathcal{L}^{\Sigma}_{cpa\text{-}0}$
$k \leftarrow \Sigma.Gen$
$\underline{EAVESDROP(m_0, m_1 \in \Sigma.\mathcal{M}):}$
$c := \Sigma.Enc(k, m_0)$
$return \;\; c$

$\mathcal{L}^{\Sigma}_{cpa\text{-}1}$
$k \leftarrow \Sigma.Gen$
$\underline{EAVESDROP(m_0, m_1 \in \Sigma.\mathcal{M}):}$
$c := \Sigma.Enc(k, m_1)$
$return \;\; c$

Observe cómo la clave k se elige en el momento de la inicialización y es estática para todas las llamadas a Enc. La seguridad CPA a menudo se llama seguridad "IND-CPA", que significa "indistinguibilidad de textos cifrados bajo ataque de texto plano elegido".

9.1. Límites del cifrado determinista

Ya hemos visto cifrados en bloque / PRPs, que parecen satisfacer todo lo necesario para un esquema de cifrado seguro. Para un cifrado en bloque, F corresponde al cifrado, F^{-1} corresponde al descifrado, y todas las salidas de F parecen pseudoaleatorias. ¿Qué más se podría pedir en un buen esquema de cifrado?

Ejemplo 9.1. Veremos que un cifrado en bloque, cuando se usa "tal cual", no es un esquema de cifrado seguro CPA. Sea F el cifrado en bloque y supongamos que su longitud de bloque es *blen*.

Considere el siguiente adversario $\mathcal{A}$, que intenta distinguir las librerías $\mathcal{L}^{\Sigma}_{\text{otint-0}}$ y $\mathcal{L}^{\Sigma}_{\text{otint-1}}$:

$\mathcal{A}$

$c_1 := \text{EAVESDROP}(0^{blen}, 0^{blen})$
$c_1 := \text{EAVESDROP}(0^{blen}, 1^{blen})$
return $c_1 \stackrel{?}{=} c_2$

$\mathcal{A}$

$c_1 := \text{EAVESDROP}(0^{blen}, 0^{blen})$
$c_1 := \text{EAVESDROP}(0^{blen}, 1^{blen})$
return $c_1 \stackrel{?}{=} c_2$

$\diamond$

$\mathcal{L}^{\Sigma}_{\text{cpa-0}}$

$k \leftarrow \{0,1\}^{\lambda}$

EAVESDROP(m_0, m_1):
$c := F(k, m_0)$
return c

Cuando $\mathcal{A}$ está vinculado a $\mathcal{L}_{\text{cpa-0}}$, el algoritmo EAVESDROP cifrará su primer argumento. Por lo tanto, c_1 y c_2 se calcularán como $F(k, 0^{blen})$. Dado que F es una función determinista, esto da como resultado salidas idénticas de EAVESDROP. En otras palabras, $c_1 = c_2$, y $\mathcal{A} \diamond \mathcal{L}_{\text{cpa-0}}$ siempre produce 1.

$\mathcal{A}$

$c_1 := \text{EAVESDROP}(0^{blen}, 0^{blen})$
$c_1 := \text{EAVESDROP}(0^{blen}, 1^{blen})$
return $c_1 \stackrel{?}{=} c_2$

$\diamond$

$\mathcal{L}^{\Sigma}_{\text{cpa-1}}$

$k \leftarrow \{0,1\}^{\lambda}$

EAVESDROP(m_0, m_1):
$c := F(k, m_1)$
return c

Cuando $\mathcal{A}$ está vinculado a $\mathcal{L}_{\text{cpa-1}}$, el algoritmo EAVESDROP cifrará su segundo argumento. Por lo tanto, c_1 y c_2 se calculan como $c_1 = F(k, 0^{blen})$ y $c_2 = F(k, 1^{blen})$. Dado que F es una permutación, $c_1 \neq c_2$, por lo que $\mathcal{A} \diamond \mathcal{L}_{\text{cpa-1}}$ nunca produce 1.

Este adversario tiene una ventaja de 1 para distinguir las librerías, por lo que el cifrado en bloque F sin modificaciones no es un esquema de cifrado seguro CPA.

9.1.1. Imposibilidad del cifrado determinista

La razón por la que un cifrado en bloque "puro" no proporciona seguridad CPA es que es **determinista**. Llamar a $\text{Enc}(k, m)$ dos veces, con la misma clave y el mismo texto plano, conduce al mismo texto cifrado. Incluso el one-time pad es determinista.[1] Uno de los primeros y más importantes aspectos de la seguridad

[1] Recuerda, siempre podemos considerar qué sucederá al ejecutar el one-time pad dos veces con la misma clave + texto plano. La definición de indistinguibilidad de un solo uso no nos da ninguna garantía de seguridad sobre el uso del one-time pad de esta manera, pero aún podemos considerarlo como un experimento mental.

CPA es que es incompatible con el cifrado determinista. **¡El cifrado determinista nunca puede ser seguro bajo CPA!** En otras palabras, podemos atacar la seguridad CPA de cualquier esquema Σ, sabiendo solo que tiene cifrado determinista. El ataque es una simple generalización de nuestro ataque contra un PRP puro:

$\mathcal{A}$

Elegir arbitrariamente textos planos distintos $x, y \in \Sigma.\mathcal{M}$

$c_1 := \text{EAVESDROP}(x, x)$
$c_1 := \text{EAVESDROP}(x, y)$
return $c_1 \stackrel{?}{=} c_2$

Una buena manera de pensar en lo que sale mal con el cifrado determinista es que **revela si dos textos cifrados codifican el mismo texto plano**, y esto no está permitido por la seguridad CPA. Piensa en sobres sellados como una analogía para el cifrado. ¡No debería poder decir si dos sobres sellados contienen el mismo texto! Solo ahora estamos viendo este problema porque esta es la primera vez que nuestra definición de seguridad permite que un adversario vea múltiples textos cifrados encriptados con la misma clave.

9.1.2. Evitando el cifrado determinista

¿Es siquiera posible la seguridad CPA? ¿Cómo podemos crear exactamente un esquema de cifrado no determinista? ¡Esto suena desafiante! Debemos diseñar un algoritmo Enc tal que llamarlo dos veces con el mismo texto plano y la misma clave resulte en textos cifrados diferentes (de lo contrario, el ataque $\mathcal{A}$ anterior viola la seguridad CPA). ¡Además, debe ser posible descifrar todos esos cifrados diferentes del mismo texto plano al valor correcto!

Hay 3 formas generales de diseñar un esquema de cifrado que no sea determinista:

- El cifrado/descifrado puede tener **estado**, lo que significa que cada llamada a Enc o Dec modificará realmente el valor de k. Un inconveniente importante del cifrado con estado es que la sincronización entre el remitente y el receptor es frágil y puede romperse si se pierde un texto cifrado en tránsito.
- El cifrado puede ser **aleatorizado**. Cada vez que se cifra un texto plano, el algoritmo Enc elige aleatoriedad fresca e independiente específica para ese cifrado. El principal desafío en el diseño de un método de cifrado aleatorizado es incorporar la aleatoriedad en cada texto cifrado de tal manera que el descifrado siga siendo posible. Aunque esto suena bastante desafiante, ya hemos visto un método de este tipo y demostraremos su seguridad CPA en las próximas secciones. En este curso nos centraremos casi por completo en el cifrado aleatorizado.
- El cifrado puede basarse en **nonce**. Un "nonce" es un "número usado solo una vez", y se refiere a un argumento adicional que se pasa a los algoritmos Enc y Dec. Un nonce no necesita ser elegido aleatoriamente; no necesita ser secreto; solo necesita ser **distinto** entre todas las llamadas realizadas a Enc. Al garantizar que alguna entrada a Enc sea diferente cada vez (incluso cuando se repiten la clave y el texto plano), el algoritmo Enc puede ser determinista y aún así proporcionar seguridad CPA.

El cifrado basado en nonce requiere un cambio en la interfaz del cifrado y, por lo tanto, también un cambio en las definiciones de corrección y seguridad. La sintaxis de los algoritmos de cifrado/descifrado se actualiza a $\text{Enc}(k, v, m)$ y $\text{Dec}(k, v, c)$, donde v es un nonce. La propiedad de corrección es que $\text{Dec}(k, v, \text{Enc}(k, v, m)) = m$ para todo k, v, m, por lo que tanto el algoritmo de cifrado como el de descifrado deben usar el mismo nonce. La definición de seguridad permite que el adversario elija el nonce, pero da un error si el adversario intenta cifrar múltiples textos cifrados con el mismo nonce. De esta manera, la definición exige que los nonces sean distintos.

$k \leftarrow \Sigma.\text{Gen}$ $V := \emptyset$ EAVESDROP(v, $m_0, m_1 \in \Sigma.\mathcal{M}$): if $v \in V$: return error $V := V \cup \{v\}$ $c := \Sigma.\text{Enc}(v, k, m_0)$ return c	$\approx$	$k \leftarrow \Sigma.\text{Gen}$ $V := \emptyset$ EAVESDROP(v, $m_0, m_1 \in \Sigma.\mathcal{M}$): if $v \in V$: return error $V := V \cup \{v\}$ $c := \Sigma.\text{Enc}(v, k, m_1)$ return c

Ten en cuenta que el programa que llama estas librerías proporciona un solo valor v (no un v_0 y v_1). Ambas librerías usan el nonce v que se proporciona, y esto implica que el esquema de cifrado no necesita ocultar v. Si algo es lo mismo entre ambas librerías, entonces no es necesario ocultarlo para que las librerías sean indistinguibles.

Si un esquema de cifrado no se incluye en una de estas tres categorías, no puede satisfacer nuestra definición de seguridad CPA. Puedes y debes usar el cifrado determinista como una comprobación de cordura contra cualquier algoritmo de cifrado propuesto.

9.2. Textos cifrados pseudoaleatorios

Cuando introdujimos la seguridad de un solo uso (en la Sección 5.4), teníamos dos variantes de la definición. La variante más general decía, a grandes rasgos, que los cifrados de m_0 deberían verse como cifrados de m_1. La variante más específica decía que los cifrados de cada m deberían verse uniformes.

Podemos hacer algo similar para la seguridad CPA, enunciando una definición de seguridad que diga "los cifrados de m se ven uniformes". Tenga en cuenta que no es suficiente usar las mismas librerías de seguridad de la definición de seguridad de un solo uso. Es importante que la librería permita múltiples cifrados bajo la misma clave. El hecho de que un solo cifrado sea pseudoaleatorio no significa que múltiples cifrados parezcan conjuntamente pseudoaleatorios. En particular, pueden no parecer independientes (este fue un problema que vimos al discutir la dificultad de construir una PRF a partir de un PRG).

Definición 9.2.1 (Seguriad pCPA). *Sea Σ un esquema de cifrado. Decimos que Σ tiene* ***textos cifrados pseudoaleatorios en presencia de ataques de texto plano elegido (seguridad pCPA)*** *si $\mathcal{L}^{\Sigma}_{pCPA} \approx \mathcal{L}^{\Sigma}_{pCPA\text{-}\$}$, donde:*

$\mathcal{L}^{\Sigma}_{pCPA}$
$k \leftarrow \Sigma.Gen$

$\underline{CTXT(m \in \Sigma.\mathcal{M}):}$
$c := \Sigma.Enc(k, m)$
return c

$\mathcal{L}^{\Sigma}_{pCPA\text{-}\$}$
$\underline{CTXT(m \in \Sigma.\mathcal{M}):}$
$c \leftarrow \Sigma.\mathcal{C}$
return c

Esta definición será útil porque:

- Es más fácil probar la seguridad pCPA que probar la seguridad CPA. Las pruebas para la seguridad CPA tienden a ser aproximadamente el doble de largas y el doble de repetitivas, ya que implican llegar a una "librería a mitad de camino" y luego realizar la misma secuencia de pasos intermedios a la inversa. Realizar la prueba solo hasta el mismo punto intermedio generalmente es suficiente para probar la seguridad pCPA.
- La seguridad pCPA implica la seguridad CPA. Lo mostramos a continuación, pero la idea principal es la misma que en el caso de la seguridad de un solo uso. Si los cifrados de todos los textos planos se ven uniformes, entonces los cifrados de m_0 se ven como los cifrados de m_1.
- La mayoría de los esquemas que consideraremos logran pCPA de todos modos.

Aun así, la mayor parte de nuestra discusión de alto nivel sobre las propiedades de seguridad se basará en la seguridad CPA. Es la definición "mínima" (es decir, la menos restrictiva) que parece capturar nuestras intuiciones de seguridad.

Afirmación 9.2.2. *Si un esquema de cifrado tiene seguridad pCPA, entonces también tiene seguridad CPA.*

Demostración. Queremos probar que $\mathcal{L}^{\Sigma}_{\text{cpa-0}} \approx \mathcal{L}^{\Sigma}_{\text{cpa-1}}$ usando la suposición de que $\mathcal{L}^{\Sigma}_{\text{pCPA}} \approx \mathcal{L}^{\Sigma}_{\text{pCPA-\$}}$.

La secuencia de librerías es la siguiente:

$\mathcal{L}^{\Sigma}_{\text{cpa-0}}$:

$\mathcal{L}^{\Sigma}_{\text{cpa-0}}$
$k \leftarrow \Sigma.\text{Gen}$

$\underline{\text{EAVESDROP}(m_0, m_1):}$
$c := \Sigma.\text{Enc}(k, m_0)$
return c

El punto de partida es $\mathcal{L}^{\Sigma}_{\text{cpa-0}}$, como se esperaba.

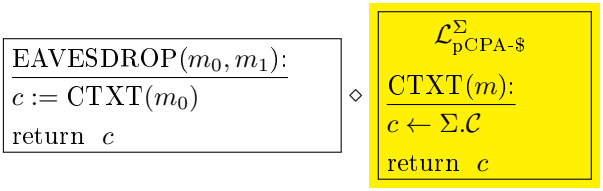
$\underline{\text{EAVESDROP}(m_0, m_1):}$
$c :=$ CTXT(m_0)
return c

$\diamond$

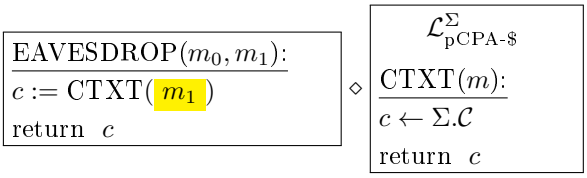
$\mathcal{L}^{\Sigma}_{\text{pCPA}}$
$k \leftarrow \Sigma.\text{Gen}$

$\underline{\text{CTXT}(m):}$
$c := \Sigma.\text{Enc}(k, m)$
return c

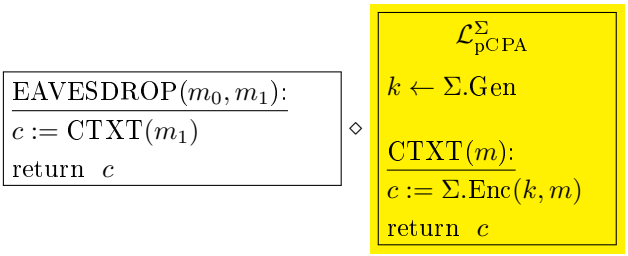
Puede parecer extraño, pero hemos factorizado aún más la llamada a `Enc` en una subrutina. Parece que todo de $\mathcal{L}^{\Sigma}_{\text{cpa-0}}$ ha sido factorizado, pero en realidad la librería original todavía "toma la decisión" de cuál de m_0, m_1 cifrar.



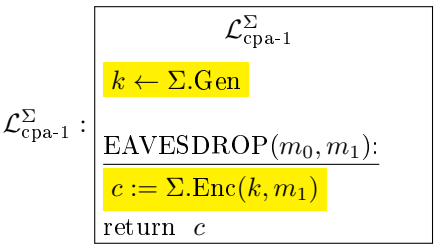
Hemos reemplazado $\mathcal{L}^{\Sigma}_{\text{pCPA}}$ con $\mathcal{L}^{\Sigma}_{\text{pCPA-\$}}$. Por nuestra suposición, el cambio es indistinguible.



Hemos cambiado el argumento que se pasa a CTXT. Esto no tiene ningún efecto en el comportamiento de la librería, ya que CTXT ignora por completo su argumento en estas librerías.



La imagen especular de un paso anterior; reemplazamos $\mathcal{L}^{\Sigma}_{\text{pCPA-\$}}$ con $\mathcal{L}^{\Sigma}_{\text{pCPA}}$.



La librería $\mathcal{L}^{\Sigma}_{\text{pCPA}}$ se ha integrado y el resultado es $\mathcal{L}^{\Sigma}_{\text{cpa-1}}$.

La secuencia de librerías muestra que $\mathcal{L}^{\Sigma}_{\text{cpa-0}} \approx \mathcal{L}^{\Sigma}_{\text{cpa-0}}$, como se deseaba.

■

9.3. Cifrado seguro bajo CPA basado en PRFs

La seguridad CPA presenta un desafío significativo; sus objetivos parecen difíciles de conciliar. Por un lado, necesitamos un método de cifrado que sea aleatorizado, de modo que cada texto plano m se mapee a una gran cantidad de posibles textos cifrados. Por otro lado, el método de descifrado debe ser capaz de reconocer todos estos diversos textos cifrados como cifrados de m.

Sin embargo, ¡ya hemos visto una manera de hacer esto! En el Capítulo 8 motivamos el concepto de una PRF (Función Pseudoaleatoria) con la siguiente técnica de cifrado. Si Alice y Bob comparten una enorme tabla T inicializada con datos uniformes, entonces Alice puede cifrar un texto plano m para Bob diciendo algo como "esto está cifrado con un one-time pad, usando la clave #674696273" y enviando $T[674696273] \oplus m$. Ver el número 674696273 no ayuda al espía a saber qué es $T[674696273]$. Una PRF permite a Alice y Bob hacer el mismo cifrado mientras comparten solo una clave corta k. En lugar de la enorme tabla T, pueden usar

una PRF $F(k, \cdot)$ para derivar un valor pseudoaleatorio común. Conocer un valor r no ayuda al adversario a predecir $F(k, r)$, cuando k es secreto.

Por lo tanto, traducido a una notación PRF más precisa, un cifrado de m se verá como $(r, F(k, r) \oplus m)$. Dado que Bob también tiene k, puede descifrar cualquier texto cifrado de esta forma calculando $F(k, r)$ y haciendo XOR con el segundo componente del texto cifrado para recuperar m.

Queda por decidir exactamente cómo Alice elegirá los valores r. Argumentamos, informalmente, que mientras estos valores r no se repitan, la seguridad se conserva. Esto es definitivamente cierto, y la distinción de los valores r es crítica. Recordemos que hay 3 formas de evitar el cifrado determinista, y las 3 funcionarían aquí:

- En un cifrado con estado, r podría usarse como un contador. Se usa $r = i$ para cifrar/descifrar el i-ésimo texto cifrado.
- En un cifrado aleatorizado, se elige r uniformemente al azar para cada cifrado. Si los valores r son cadenas suficientemente largas, entonces repetir un valor r debería ser despreciablemente probable.
- En un cifrado basado en *nonce*, podemos simplemente dejar que r sea el *nonce*. En el contexto basado en *nonce*, está garantizado que estos valores no se repetirán.

En esta sección mostraremos la prueba de seguridad para el caso del cifrado aleatorizado, ya que es el contexto más tradicional y también algo más robusto que los otros.

Sea F una PRF segura con entrada de longitud λ. Definimos el siguiente esquema de cifrado basado en F:

Σ con seguridad CPA basada en PRF:

$\mathcal{K} = \{0,1\}^\lambda$	Gen:	Enc(k, m):	Dec$(k, (r, x))$:
$\mathcal{M} = \{0,1\}^{\text{out}}$	$k \leftarrow \{0,1\}^\lambda$	$r \leftarrow \{0,1\}^\lambda$	$m := F(k, r) \oplus x$
$\mathcal{C} = \{0,1\}^\lambda \times \{0,1\}^{\text{out}}$	return k	$x := F(k, r) \oplus m$	return m
		return (r, x)	

Es fácil comprobar que el esquema satisface la propiedad de corrección.

Afirmación 9.3.1. *El esquema basado en PRF definido anteriormente tiene seguridad pCPA (y por lo tanto seguridad CPA) si F es una PRF segura.*

La prueba tiene más pasos que otras pruebas que hemos visto antes, y algunos pasos son sutiles. Así que usemos un diálogo socrático para ilustrar la estrategia detrás de la prueba:

Salviati: Los textos cifrados del esquema son indistinguibles de la aleatoriedad uniforme.

Simplicio: ¡Salviati, hablas con tanta confianza! Dime por qué dices que estos textos cifrados parecen pseudoaleatorios.

Salviati: ¡Simple! Los textos cifrados tienen la forma $(r, F(k, r) \oplus m)$. Por su propia definición, r se elige uniformemente, mientras que $F(k, r) \oplus m$ es como un texto cifrado de one-time pad que también está distribuido uniformemente.

SIMPLICIO: Tu afirmación sobre r es evidente, pero $F(k, r) \oplus m$ me confunde. Esto no se parece al one-time pad que hemos discutido. Para empezar, la misma k se usa "cada vez", no "una sola vez".

SALVIATI: Dije que era simplemente "como" un one-time pad. La "clave" del one-time pad no es k sino $F(k, r)$. Y dado que F es una función pseudoaleatoria, todas sus salidas parecerán independientemente uniformes (por no mencionar que no están correlacionadas con sus respectivos r), incluso cuando se usa la misma semilla cada vez. ¿No es esto lo que requerimos de una clave de one-time pad?

SIMPLICIO: Ya veo, pero seguramente las salidas de F parecen independientes solo cuando sus entradas son distintas, ¿no? Sé que F es determinista, y esto puede llevar a que se use la misma "clave de one-time pad" en diferentes ocasiones.

SALVIATI: Tu escepticismo te sirve bien en este empeño, Simplicio. De hecho, el núcleo de tu preocupación es que Alice pueda elegir r de tal manera que se repita. Yo digo que esto es despreciablemente probable, por lo que podemos ignorar con seguridad un evento tan molesto.

SIMPLICIO: Molesto, de hecho, pero ¿por qué dices que es poco probable que r se repita?

SALVIATI: ¡Oh, Simplicio, ahora te estás volviendo molesto! Este valor r tiene λ bits de longitud y se elige uniformemente al azar cada vez. ¿No recuerdas nuestra dolorosamente larga discusión sobre la paradoja del cumpleaños?

SIMPLICIO: Oh, sí, ahora lo recuerdo bien. Ahora creo que entiendo todo tu razonamiento: en todos los textos cifrados que se generan, es poco probable que r se repita debido a la paradoja del cumpleaños. Ahora, dado que r nunca se repite, Alice invoca la PRF con entradas distintas. Una PRF invocada con entradas distintas proporciona salidas que son uniformemente aleatorias para todos los efectos. Por lo tanto, usar estas salidas como one-time pads oculta completamente el texto plano. ¿Es correcto, Salviati?

SALVIATI: ¡Excelente! Ahora podemos volver a discutir el movimiento del Sol y la Tierra...

Busca que el resumen final de Simplicio se refleje en la secuencia de librerías utilizadas en la prueba formal:

Demostración. Probamos que $\mathcal{L}^{\Sigma}_{\text{pCPA}} \approx \mathcal{L}^{\Sigma}_{\text{pCPA-\$}}$ usando la siguiente secuencia de librerías:

$\mathcal{L}^{\Sigma}_{\text{pCPA}}$:

$\mathcal{L}^{\Sigma}_{\text{pCPA}}$
$k \leftarrow \Sigma.\text{Gen}$
CTXT(m):
$r \leftarrow \{0,1\}^{\lambda}$
$x := F(k, r) \oplus m$
return (r, x)

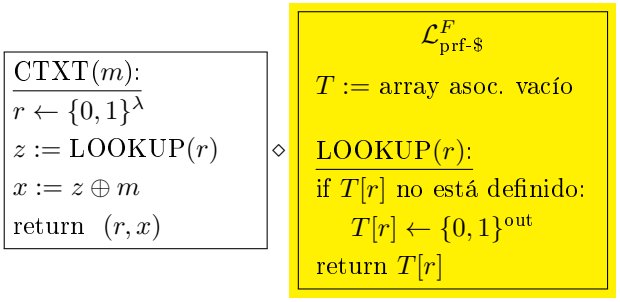
El punto de partida es $\mathcal{L}^{\Sigma}_{\text{pCPA}}$. Los detalles de Σ se han completado y resaltado.

CTXT(m):
$r \leftarrow \{0,1\}^{\lambda}$
$z :=$ LOOKUP(r)
$x := z \oplus m$
return (r, x)

$\diamond$

$\mathcal{L}^{F}_{\text{prf}}$
$k \leftarrow \{0,1\}^{\lambda}$
LOOKUP(r):
return $F(k, r)$

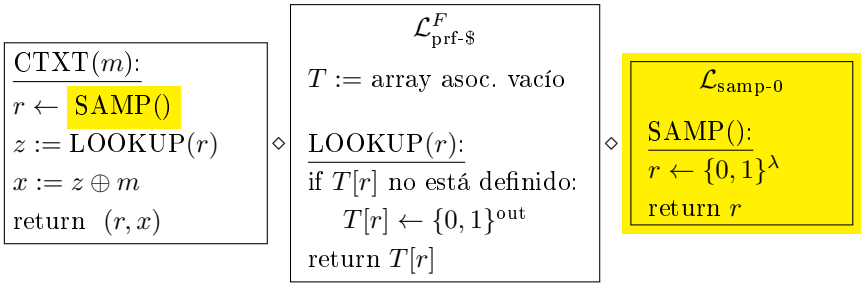
Las afirmaciones relativas a la PRF se han factorizado en términos de la librería $\mathcal{L}^{F}_{\text{prf}}$.



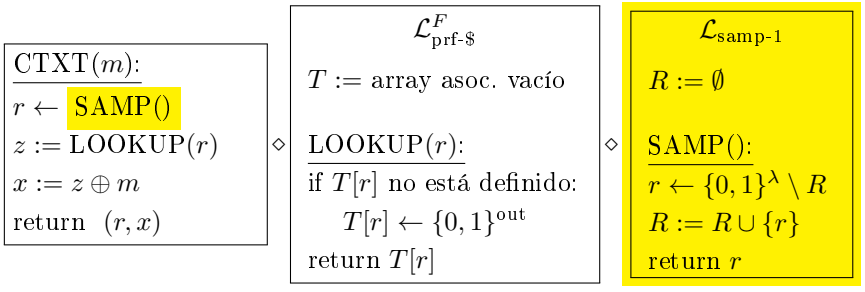
Hemos reemplazado $\mathcal{L}^F_{\text{prf}}$ con $\mathcal{L}^F_{\text{prf-\$}}$. De la seguridad PRF de F, estas dos librerías son indistinguibles.

En este punto de la prueba, es fácil imaginar que hemos terminado. Los textos cifrados tienen la forma (r, x), donde r se elige uniformemente y x es el resultado de cifrar el texto plano con lo que parece ser un one-time pad. Sin embargo, mirando con más cuidado, la "clave del one-time pad" es $F(k, r)$ - ¡un valor que podría usarse más de una vez si r se repite alguna vez! Como Simplicio señaló correctamente, una PRF proporciona salidas (con apariencia de) aleatorias independientes cuando se llama con entradas distintas. ¡Pero en nuestra librería actual no hay garantía de que las entradas de la PRF sean distintas! Nuestra prueba debe contener explícitamente un razonamiento sobre por qué es poco probable que las entradas de la PRF se repitan. Lo hacemos apelando al lema de muestreo con reemplazo del Lema 6.4.3.

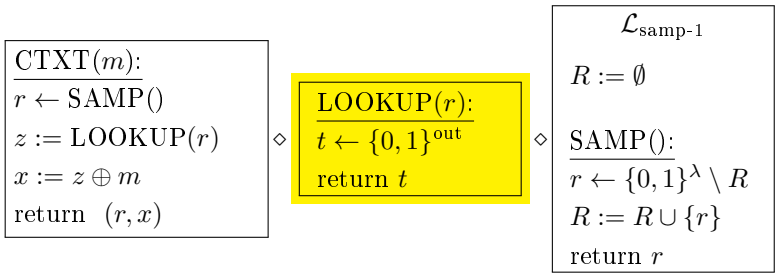
Primero factorizamos el muestreo de valores r en una subrutina. La subrutina corresponde a la librería $\mathcal{L}_{\text{samp-0}}$ del Lema 6.4.3:



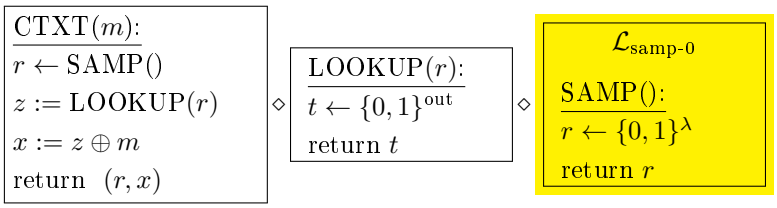
A continuación, $\mathcal{L}_{\text{samp-0}}$ se reemplaza por $\mathcal{L}_{\text{samp-1}}$. Según el Lema 6.4.3, la diferencia es indistinguible:



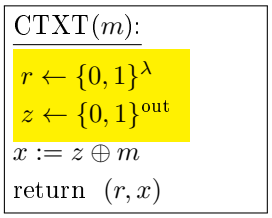
Inspeccionando la librería anterior, podemos razonar que los argumentos a LOOKUP tienen la garantía de no repetirse nunca. Por lo tanto, la librería $\mathcal{L}^F_{\text{prf-\$}}$ se puede simplificar enormemente. En particular, la condición if en $\mathcal{L}^F_{\text{prf-\$}}$ siempre es verdadera. La simplificación no tiene ningún efecto en el comportamiento de salida de la librería:



Ahora, de hecho, estamos usando one-time pads únicos para enmascarar el texto plano. Estamos en mucho mejor forma que antes. Recordemos que nuestro objetivo es llegar a una librería en la que las salidas de CTXT se elijan uniformemente. Estas salidas incluyen el valor r, *¡pero ahora r ya no se elige uniformemente!* Debemos revertir r para que se muestree uniformemente, y entonces estaremos cerca de la línea de meta.

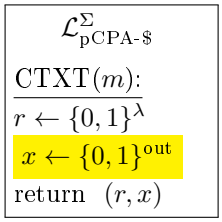


Como se prometió, $\mathcal{L}_{\text{samp-1}}$ ha sido reemplazado por $\mathcal{L}_{\text{samp-0}}$. La diferencia es indistinguible debido al Lema 6.4.3.



Todas las llamadas a subrutinas se han integrado; sin efecto en el comportamiento de salida de la librería.

$\mathcal{L}^{\Sigma}_{\text{pCPA-\$}}$:



Hemos aplicado la regla del one-time pad con respecto a las variables z y x, pero omitimos los pasos muy familiares (factorizar, reemplazar librería, integrar) que hemos visto varias veces antes. La librería resultante es precisamente $\mathcal{L}^{\Sigma}_{\text{pCPA-\$}}$ ya que muestrea uniformemente de $\Sigma.\mathcal{C} = \{0,1\}^{\lambda} \times \{0,1\}^{\text{out}}$.

La secuencia de híbridos muestra que $\mathcal{L}^{\Sigma}_{\text{pCPA}} \approx \mathcal{L}^{\Sigma}_{\text{pCPA-\$}}$, por lo que Σ tiene textos cifrados pseudoaleatorios. ■

Capítulo 10

Modos de operación de cifrados en bloque

Una de las desventajas del esquema de cifrado seguro bajo CPA anterior es que sus textos cifrados son λ bits más largos que sus textos planos. En el caso común en el que estamos usando un cifrado en bloque con una longitud de bloque $blen = \lambda$, esto significa que los textos cifrados son el doble de largos que los textos planos. ¿Existe alguna forma de cifrar datos (especialmente muchos) sin requerir una sobrecarga tan significativa?

Un modo de operación de cifrado en bloque se refiere a una forma de usar un cifrado en bloque para cifrar eficientemente una gran cantidad de datos (más de un solo bloque). En este capítulo, veremos los modos más comunes para el cifrado seguro bajo CPA de textos planos largos.

10.1. Un recorrido por los modos comunes

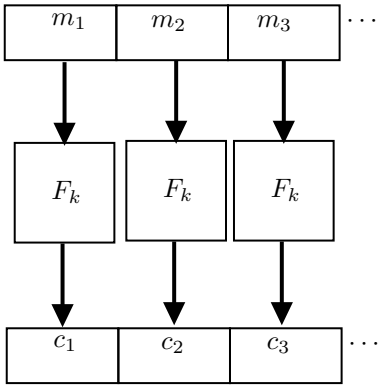
Como de costumbre, *blen* denotará la longitud del bloque de un cifrado en bloque F. En nuestros diagramas, escribiremos F_k como abreviatura de $F(k, \cdot)$. Cuando m es el texto plano, escribiremos $m = m_1||m_2||\cdots||m_\ell$, donde cada m_i es un solo bloque (por lo que ℓ es la longitud del texto plano medida en bloques). Por ahora, asumiremos que m es un múltiplo exacto de la longitud del bloque.

10.1.1. ECB: Electronic codebook (¡NUNCA, NUNCA USES ESTO! ¡NUNCA!)

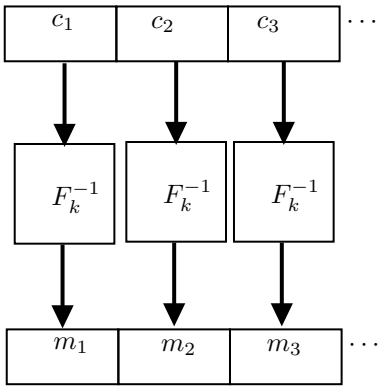
La forma más obvia de usar un cifrado en bloque para cifrar un mensaje largo es simplemente aplicar el cifrado en bloque independientemente a cada bloque. La única razón para conocer este modo es saber que nunca se debe usar. No se puede decir suficientes veces: ¡nunca uses el modo ECB! No proporciona seguridad de cifrado; ¿puedes ver por qué?

Modo ECB:

$\underline{\text{Enc}(k, m_1||m_2||\cdots||m_\ell)}$
for $i = 1$ to ℓ
  $c_i := F(k, m_i)$
return $c_1||\cdots||c_\ell$



$\underline{\text{Dec}(k, c_1||c_2||\cdots||c_\ell)}$
for $i = 1$ to ℓ
  $m_i := F^{-1}(k, c_i)$
return $m_1||\cdots||m_\ell$

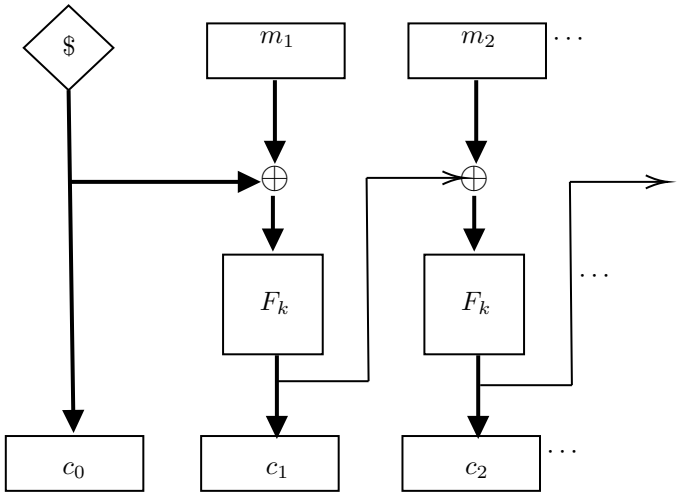


10.1.2. CBC: Encadenamiento de bloques cifrados

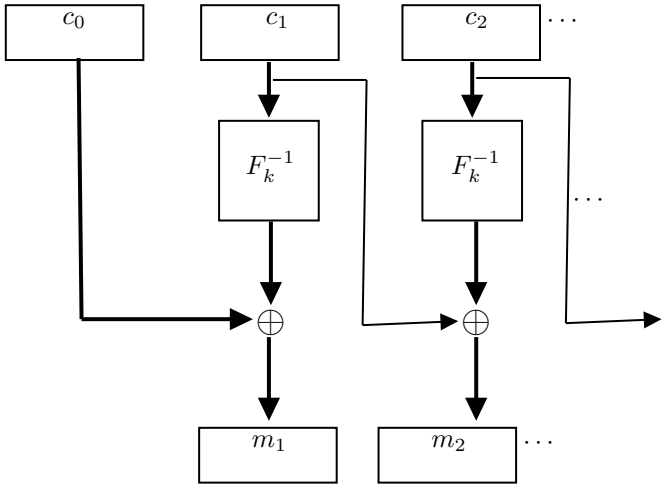
CBC (que significa encadenamiento de bloques cifrados) es el modo más común en la práctica. El cifrado CBC de un texto plano de ℓ bloques tiene una longitud de $\ell+1$ bloques. El primer bloque del texto cifrado se llama vector de inicialización (IV). Aquí hemos descrito el modo CBC como un cifrado aleatorizado, donde el IV de cada texto cifrado se elige uniformemente. Como ya sabes, la aleatorización es necesaria (pero no suficiente) para lograr la seguridad CPA, y de hecho el modo CBC proporciona seguridad CPA.

Modo CBC:

$\underline{\text{Enc}(k, m_1||m_2||\cdots||m_\ell)}$
$c_0 \leftarrow \{0,1\}^{blen}$
for $i = 1$ to ℓ
  $c_i := F(k, m_i \oplus c_{i-1})$
return $c_0||c_1||\cdots||c_\ell$



$\underline{\text{Dec}(k, c_0||c_1||\cdots||c_\ell)}$
for $i = 1$ to ℓ
  $m_i := F^{-1}(k, c_i) \oplus c_{i-1}$
return $m_1||\cdots||m_\ell$



10.1.3. CTR: Contador

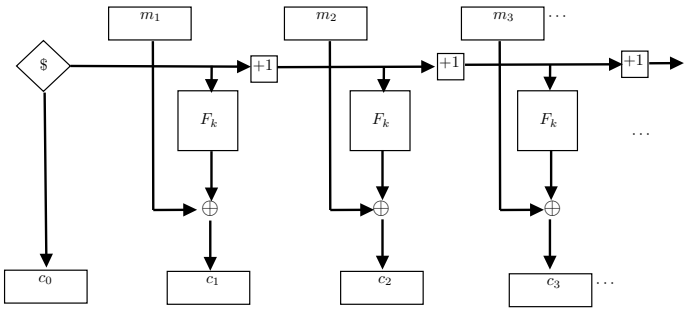
El siguiente modo más común en la práctica es el modo contador (generalmente abreviado como modo CTR). Al igual que el modo CBC, implica un bloque IV adicional r que se elige uniformemente. La idea es utilizar la secuencia

$$F(k, r); \quad F(k, r+1); \quad F(k, r+2); ...$$

como un one-time pad largo para enmascarar el texto plano. Dado que r es un bloque de bits, las expresiones de suma como $r+1$ se refieren a la suma módulo 2^{blen} (este es el comportamiento típico de la suma sin signo en un procesador).

Modo CTR:

$\underline{\text{Enc}(k, m_1||m_2||\cdots||m_\ell)}$
$r \leftarrow \{0,1\}^{blen}$
$c_0 := r$
for $i = 1$ to ℓ
  $c_i := F(k,r) \oplus m_i$
  $r := r+1 \mod 2^{blen}$
return $c_0||c_1||\cdots||c_\ell$



10.1.4. OFB: Retroalimentación de salida

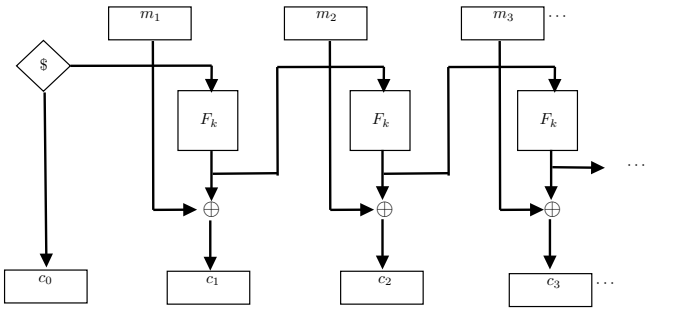
El modo OFB (retroalimentación de salida) rara vez se usa en la práctica. Lo incluiremos en nuestra discusión porque tiene la prueba de seguridad más sencilla. Al igual que con los modos CBC y CTR, OFB comienza con un IV aleatorio r, y luego usa la secuencia:

$$F(k,r); \quad F(k,F(k,r)); \quad F(k,F(k,F(k,r)));...$$

como un one-time pad para enmascarar el texto plano.

Modo OFB:

$\underline{\text{Enc}(k, m_1||m_2||\cdots||m_\ell)}$
$r \leftarrow \{0,1\}^{blen}$
$c_0 := r$
for $i = 1$ to ℓ
  $r := F(k,r)$
  $c_i := r \oplus m_i$
return $c_0||c_1||\cdots||c_\ell$



10.1.5. Comparar y contrastar

Los modos CBC y CTR son esencialmente los únicos dos modos que se consideran en la práctica para la seguridad CPA. Ambos proporcionan las mismas garantías de seguridad, por lo que cualquier comparación entre los dos debe basarse en factores ajenos a la definición de seguridad CPA. Aquí hay algunas propiedades que a menudo se consideran al elegir entre estos modos:

- Aunque no hemos mostrado el algoritmo de descifrado para el modo CTR, ni siquiera usa la inversa del cifrado en bloque F^{-1}. Esto es similar a nuestro esquema de cifrado basado en PRF del capítulo anterior (de hecho, el modo CTR se reduce a esa construcción cuando se restringe a textos planos de 1 bloque). Estrictamente hablando, esto significa que el modo CTR se puede instanciar a partir de una

PRF; no necesita una PRP. Sin embargo, en la práctica es raro encontrar una PRF eficiente que no sea una PRP.

- El cifrado en modo CTR se puede paralelizar. Una vez que se ha elegido el IV, el i-ésimo bloque del texto cifrado se puede calcular sin calcular primero los $i-1$ bloques anteriores. El modo CBC no tiene esta propiedad, ya que es inherentemente secuencial. Sin embargo, ambos modos tienen un algoritmo de descifrado paralelizable.
- Si las llamadas al cifrado en bloque son costosas, podría ser deseable precalcularlas y almacenarlas antes de que se conozca el texto plano. El modo CTR permite esto, ya que solo el IV afecta la entrada dada al cifrado en bloque. En el modo CBC, el texto plano influye en las entradas al cifrado en bloque, por lo que estas llamadas no se pueden precalcular antes de que se conozca el texto plano.
- Es relativamente fácil modificar CTR para admitir textos planos que no son un múltiplo exacto de la longitud del bloque. El modo CBC también admite tales textos planos, pero está lejos de ser trivial.
- Hasta ahora, todas las comparaciones han favorecido al modo CTR, por lo que aquí hay una propiedad importante que favorece al modo CBC. Es común que los implementadores malinterpreten las implicaciones de seguridad del IV en estos modos. Muchas implementaciones descuidadas permiten que se reutilice un IV. Técnicamente hablando, reutilizar un IV (que no sea por accidente, como lo permite el límite del cumpleaños) significa que el esquema no se implementó correctamente. Pero en lugar de culpar al desarrollador, es una buena práctica de diseño anticipar posibles usos indebidos de un sistema y, cuando sea posible, tratar de que no sean catastróficos.
- Los efectos de la reutilización del IV en el modo CTR son bastante devastadores para la privacidad del mensaje. En el modo CBC, reutilizar un IV puede ser realmente seguro, ¡si los dos textos planos tienen diferentes primeros bloques!

10.2. Seguridad CPA y textos planos de longitud variable

Aquí hay una gran sorpresa: ninguno de estos modos de cifrado en bloque logra la seguridad CPA, o al menos la seguridad CPA tal como la hemos estado definiendo.

Ejemplo 10.1. Considera un cifrado en bloque con $blen = \lambda$, utilizado en modo CBC. Como verás, no hay nada particularmente específico del modo CBC, y las mismas observaciones se aplican a los otros modos. En el modo CBC, un texto plano que consta de ℓ bloques se cifra en un texto cifrado de $\ell + 1$ bloques. En otras palabras, el texto cifrado revela el número de bloques en el texto plano. Podemos aprovechar esta observación en el siguiente ataque:

$\mathcal{A}$

$c := \text{EAVESDROP}(0^\lambda, 0^{2\lambda})$
return $|c| \stackrel{?}{=} 2\lambda$

El distinguidor elige un texto plano de 1 bloque y un texto plano de 2 bloques. Si este distinguisher está vinculado a $\mathcal{L}_{\text{cpa}-0}$, se cifra el texto plano de 1 bloque y el texto cifrado resultante tiene 2 bloques (2λ bits) de longitud. Si el distinguidor está vinculado a $\mathcal{L}_{\text{cpa-1}}$, se cifra el texto plano de 2 bloques y el texto cifrado

resultante tiene 3 bloques (3λ bits) de longitud. Simplemente verificando la longitud del texto cifrado, este distinguidor puede notar la diferencia y lograr una ventaja de 1.

Entonces, técnicamente hablando, estos modos de cifrado en bloque no proporcionan seguridad CPA, ya que los textos cifrados revelan la longitud (medida en bloques) del texto plano. Pero supongamos que realmente no nos importa ocultar la longitud de los textos planos.[1] ¿Existe una forma de hacer una definición de seguridad que diga: los textos cifrados ocultan todo sobre el texto plano, excepto su longitud?

Del ejemplo anterior, queda claro que un distinguidor puede distinguir con éxito las librerías CPA si realiza una consulta EAVESDROP(m_0, m_1) con $|m_0| \neq |m_1|$. Una forma sencilla de cambiar la definición de seguridad CPA es simplemente prohibir este tipo de consulta. Las librerías darán un mensaje de error si $|m_0| \neq |m_1|$. Esto permitiría al adversario hacer que los textos planos de desafío difieran en cualquier forma de su elección, excepto en su longitud. Realmente no importa si $|m|$ se refiere a la longitud del texto plano en bits o en bloques, lo que tenga más sentido para un esquema en particular.

A partir de ahora, cuando hablemos de esquemas de cifrado que admiten textos planos de longitud variable, la seguridad CPA se referirá a las siguientes librerías actualizadas:

$\mathcal{L}^{\Sigma}_{\text{cpa-0}}$

$k \leftarrow \Sigma.\text{Gen}$

EAVESDROP$(m_0, m_1 \in \Sigma.\mathcal{M})$:
If $|m_0| \neq |m_1|$ return error
$c := \Sigma.\text{Enc}(k, m_0)$
return c

$\mathcal{L}^{\Sigma}_{\text{cpa-1}}$

$k \leftarrow \Sigma.\text{Gen}$

EAVESDROP$(m_0, m_1 \in \Sigma.\mathcal{M})$:
If $|m_0| \neq |m_1|$ return error
$c := \Sigma.\text{Enc}(k, m_1)$
return c

En la definición de seguridad pCPA (textos cifrados pseudoaleatorios), la librería $\mathcal{L}_{\text{pCPA-\$}}$ responde a las consultas con respuestas uniformes. Dado que estas respuestas deben parecer textos cifrados, deben tener la longitud adecuada. Por ejemplo, para los modos discutidos en este capítulo, se espera que un texto plano de ℓ bloques se cifre en un texto cifrado de $(\ell + 1)$ bloques. Por lo tanto, según la longitud del texto plano que se proporciona, la librería debe elegir la longitud de texto cifrado adecuada. Ya estamos usando $\Sigma.\mathcal{C}$ para denotar el conjunto de posibles textos cifrados de un esquema de cifrado Σ. Así que extendamos un poco la notación y escribamos $\Sigma.\mathcal{C}(\ell)$ para denotar el conjunto de posibles textos cifrados para textos planos de longitud ℓ. Entonces, cuando se discuten esquemas de cifrado que admiten textos planos de longitud variable, la seguridad CPA se referirá a las siguientes librerías:

$\mathcal{L}^{\Sigma}_{\text{pCPA}}$

$k \leftarrow \Sigma.\text{Gen}$

CTXT$(m \in \Sigma.\mathcal{M})$:
$c := \Sigma.\text{Enc}(k, m)$
return c

$\mathcal{L}^{\Sigma}_{\text{pCPA-\$}}$

CTXT$(m \in \Sigma.\mathcal{M})$:
$c := \Sigma.\mathcal{C}(|m|)$
return c

[1] De hecho, ocultar la longitud de la comunicación (en el extremo, ocultar la existencia de la comunicación) es un problema muy difícil.

Ten en cuenta que la librería $\mathcal{L}_{\text{pCPA-\$}}$ no utiliza ninguna información sobre m aparte de su longitud. Esto refleja nuevamente la idea de que los textos cifrados no revelan nada sobre los textos planos aparte de su longitud.

Un ejercicio interesante es demostrar que, con respecto a estas definiciones de seguridad actualizadas, la seguridad pCPA implica la seguridad CPA como antes.

10.2.1. ¡No des por sentada la fuga de longitud!

Acabamos de pasar de requerir que el cifrado no revele ninguna información parcial a permitir casualmente que se filtre cierta información específica. ¡No nos apresuremos demasiado con esto!

Si realmente queremos admitir textos planos de longitud arbitraria, entonces la fuga de longitud es, de hecho, inevitable. Pero "inevitable" no significa "libre de consecuencias". Al observar solo la longitud del tráfico de cifrados en un canal de comunicación, son posibles muchos ataques graves. Aquí hay varios ejemplos:

- Al acceder a Google Maps, tu navegador recibe muchas teselas de imagen que componen el mapa que ves. Cada tesela de imagen tiene las mismas dimensiones de píxeles. Sin embargo, se comprimen para ahorrar recursos, y no todas las imágenes se comprimen tan significativamente como otras. Cada región del mundo tiene su propia "huella digital" bastante única de longitudes de teselas de imagen. Entonces, aunque el tráfico hacia y desde Google Maps esté cifrado, se filtran los tamaños de las teselas de imagen. De hecho, esto se puede utilizar para determinar la región para la que un usuario está solicitando un mapa.[2] La misma idea se aplica a las sugerencias de autocompletar en un formulario de búsqueda.
- La codificación de tasa de bits variable (VBR) es una técnica común en la codificación de audio/video. Cuando el flujo de datos transporta menos información (por ejemplo, una escena con una posición de cámara fija o una sección silenciosa de audio), se codifica a una tasa de bits más baja, lo que significa que cada unidad de tiempo se codifica en menos bits. En un flujo de video cifrado, los cambios en la tasa de bits se reflejan como cambios en la longitud del paquete. Cuando un usuario está viendo una película en Netflix o un video de YouTube (a diferencia de una transmisión de eventos en vivo), los cambios en la tasa de bits son consistentes y predecibles. Por lo tanto, es bastante sencillo determinar qué video se está viendo, incluso en una conexión cifrada, simplemente observando las longitudes de los paquetes.
- VBR también se utiliza en muchos programas de chat de voz cifrados. Los ataques a estas herramientas han ido aumentando en sofisticación. Los primeros ataques a programas de voz cifrada mostraron cómo identificar quién hablaba (de un conjunto de candidatos), simplemente observando el flujo de tamaños de texto cifrado. Los ataques posteriores pudieron determinar el idioma que se hablaba. ¡Finalmente, cuando se combinó con modelos lingüísticos sofisticados, se demostró que era posible incluso identificar palabras individuales hasta cierto punto!

Vale la pena enfatizar nuevamente que ninguno de estos ataques implica ningún intento de romper el cifrado. Los ataques se basan únicamente en el hecho de que el cifrado revela la longitud de los textos planos.

[2] http://blog.ioactive.com/2012/02/ssl-traffic-analysis-on-google-maps.html

10.3. Seguridad del modo OFB

En esta sección demostraremos que el modo OFB tiene textos cifrados pseudoaleatorios (cuando la longitud del bloque es $blen = \lambda$ bits). El cifrado y descifrado OFB utilizan la dirección directa de F, por lo que OFB proporciona seguridad incluso cuando F no es invertible. Por lo tanto, probaremos la seguridad asumiendo que F es simplemente una PRF.

Afirmación 10.3.1. *El modo OFB tiene seguridad pCPA, si su cifrado en bloque subyacente F es una PRF segura con parámetros $in = out = \lambda$.*

Demostración. La estructura general de la prueba es muy similar a la prueba utilizada para el esquema de cifrado basado en PRF en el capítulo anterior (Afirmación 9.3.1). Esto no es una coincidencia: si el modo OFB se restringe a textos planos de un solo bloque, ¡obtenemos exactamente el esquema de la Afirmación 9.3.1!

La idea es que cada bloque de texto cifrado (aparte del IV) se calcula como $c_i = r \oplus m_i$. Según la regla del one-time pad, basta con demostrar que los valores r son pseudoaleatorios independientes. Cada valor r es el resultado de una llamada a la PRF. Estas salidas de la PRF serán pseudoaleatorias independientes solo si todas las entradas a la PRF son distintas. En el modo OFB, usamos la salida r de una llamada anterior a la PRF como entrada para la siguiente, por lo que es muy poco probable que esta salida r de la PRF coincida con un valor de entrada anterior a la PRF. Para argumentar esto con mayor precisión, la prueba incluye librerías en los que r se elige sin reemplazo (antes de volver a cambiar r a un muestreo uniforme).

A continuación, se proporciona la secuencia formal de librerías:

$\mathcal{L}^{\text{OFB}}_{\text{pCPA}}$:

$\mathcal{L}^{\text{OFB}}_{\text{pCPA}}$

$k \leftarrow \{0,1\}^\lambda$

CTXT$(m_1||\cdots||m_\ell)$:
$r \leftarrow \{0,1\}^\lambda$
$c_0 := r$
for $i = 1$ to ℓ
  $r := F(k, r)$
  $c_i := r \oplus m_i$
return $c_0||c_1||\cdots||c_\ell$

El punto de partida es $\mathcal{L}^{\text{OFB}}_{\text{pCPA}}$, que se muestra aquí con los detalles de OFB completos.

CTXT$(m_1||\cdots||m_\ell)$:
$r \leftarrow \{0,1\}^\lambda$
$c_0 := r$
for $i = 1$ to ℓ
  $r :=$ LOOKUP(r)
  $c_i := r \oplus m_i$
return $c_0||c_1||\cdots||c_\ell$

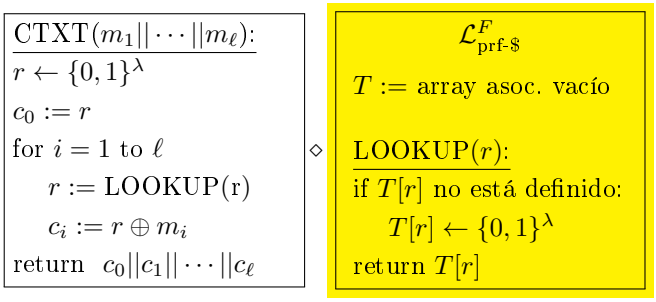
$\diamond$

$\mathcal{L}^{F}_{\text{prf}}$

$k \leftarrow \{0,1\}^\lambda$

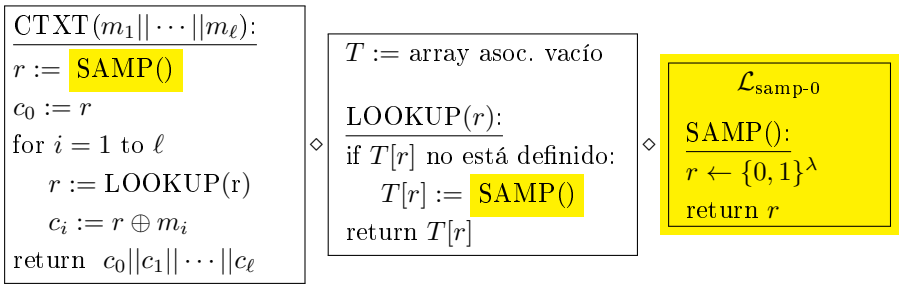
LOOKUP(r):
return $F(k, r)$

Las declaraciones pertenecientes a la PRF F se han factorizado en términos de $\mathcal{L}^{F}_{\text{prf}}$.

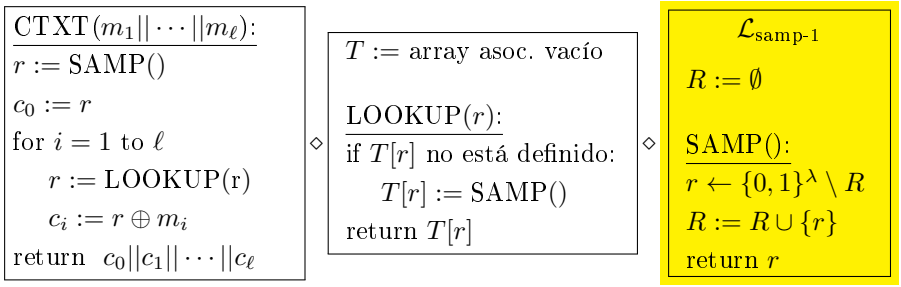


$\mathcal{L}^F_{\text{prf}}$ ha sido reemplazada por $\mathcal{L}^F_{\text{prf-\$}}$. Debido a la seguridad PRF de F, el cambio es indistinguible.

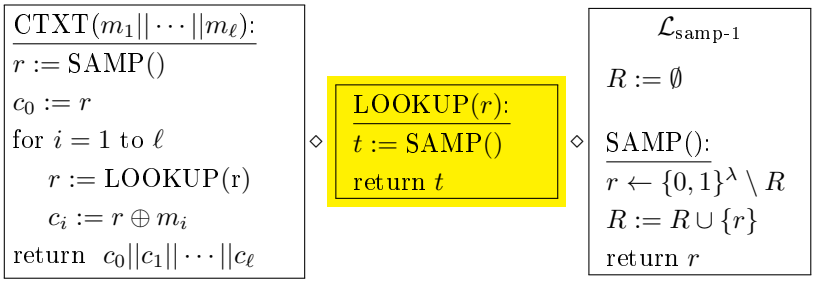
A continuación, todas las declaraciones que involucran el muestreo de valores para la variable r se factorizan en términos de la biblioteca $\mathcal{L}_{\text{samp-0}}$ del Lema 6.4.3:



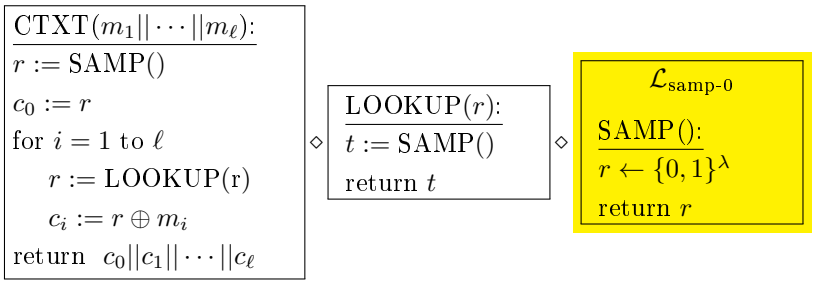
$\mathcal{L}_{\text{samp-0}}$ se reemplaza entonces por $\mathcal{L}_{\text{samp-1}}$. Según el Lema 6.4.3, este cambio es indistinguible.



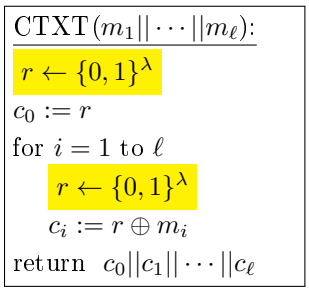
Los argumentos para LOOKUP nunca se repiten en esta librería, por lo que la librería intermedia se puede simplificar significativamente:



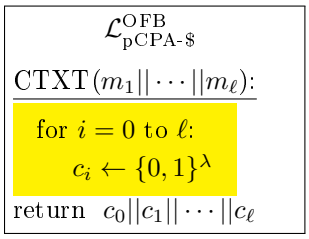
A continuación, $\mathcal{L}_{\text{samp-1}}$ se reemplaza por $\mathcal{L}_{\text{samp-0}}$. Según el Lema 6.4.3, este cambio es indistinguible.



Las llamadas a subrutinas a LOOKUP y SAMP se han integrado:



Finalmente, la regla del one-time pad se aplica dentro del bucle `for` (omitiendo algunos pasos comunes). Nótese que en la librería anterior, cada valor de r se usa solo una vez como un one-time pad. El caso $i = 0$ también se ha absorbido dentro del bucle `for`. El resultado es $\mathcal{L}^{\text{OFB}}_{\text{pCPA-\$}}$, ya que OFB cifra textos planos de ℓ bloques en $\ell + 1$ bloques.



La secuencia de librerías muestra que $\mathcal{L}^{OFB}_{\text{pCPA}} \approx \mathcal{L}^{OFB}_{\text{pCPA-\$}}$, y por lo tanto el modo OFB tiene textos cifrados pseudoaleatorios.

■

Hemos demostrado la afirmación asumiendo que F es solo una PRF, ya que el modo OFB no requiere que F sea invertible. Dado que asumimos una PRF con parámetros $in = out = \lambda$, el lema de conmutación PRP (Lema 8.4.1) muestra que OFB también es seguro cuando F es una PRP con longitud de bloque $n = \lambda$.

Capítulo 11

Texto cifrado escogido

En este capítulo discutimos las limitaciones de la definición de seguridad CPA. En resumen, la definición de seguridad CPA considera solo la información filtrada al adversario por textos cifrados generados *honestamente*. Sin embargo, no considera lo que sucede cuando a un adversario se le permite inyectar sus propios textos cifrados elaborados *maliciosamente* en un sistema honesto. Si eso sucede, entonces incluso un esquema de cifrado seguro bajo CPA puede fallar de maneras espectaculares. Comenzamos viendo un ejemplo de fallo espectacular y sorprendente, llamado un ataque de oráculo de padding:

11.1. Ataques de Oráculo de Padding

Hasta ahora hemos asumido que todos los textos planos son múltiplos exactos de la longitud del bloque. Pero los datos en el mundo real no siempre son tan complacientes. ¿Cómo se utilizan los cifradores de bloque en la práctica con datos que tienen una longitud arbitraria?

El *padding* simplemente se refiere a cualquier enfoque para codificar datos de longitud arbitraria en datos que son un múltiplo de la longitud del bloque. El único requisito es que esta codificación sea reversible. Más formalmente, un *esquema de padding* debe constar de dos algoritmos:

- `pad`: toma como entrada una cadena de cualquier longitud y produce una cadena cuya longitud es un múltiplo de la longitud del bloque.
- `unpad`: la inversa de `pad`. Requerimos que $\texttt{unpad}(\texttt{pad}(x)) = x$ para todas las cadenas x.

La idea es que el remitente puede cifrar `pad`(x), que se garantiza que es un múltiplo de la longitud del bloque; el receptor puede descifrar y ejecutar `unpad` en el resultado para obtener x.

Imagina un servidor web que recibe textos cifrados con cifrado CBC para su procesamiento. Al recibir un texto cifrado, el servidor web lo descifra con la clave apropiada y luego verifica si el texto plano tiene un padding válido.

Es importante destacar que el comportamiento observable del servidor web cambia dependiendo de si el padding es válido. Puedes imaginar que el servidor web da un mensaje de error especial en el caso de padding inválido. O, aún más ingeniosamente (pero aún realista), la diferencia en el tiempo de respuesta al procesar un texto cifrado con padding inválido es suficiente para que el ataque funcione.[1] El mecanismo para aprender la validez del padding no es importante; lo importante es simplemente el hecho de que un atacante tenga alguna forma de determinar si un texto cifrado codifica un texto plano con padding válido. No importa cómo el atacante obtenga esta información, decimos que el atacante tiene acceso a un *oráculo de padding*, que proporciona la misma información que la siguiente subrutina:

```
PADDINGORACLE(c):
m := Dec(k, c)
return VALIDPAD(m)
```

Llamamos a esto un *oráculo de padding* porque responde solo a un tipo específico de pregunta sobre la entrada. En este caso, la respuesta que da es siempre un único valor booleano.

No parece que un oráculo de padding esté filtrando información útil, y que no haya motivo de preocupación. Sorprendentemente, podemos demostrar que un atacante que no conoce la clave de cifrado k puede usar un oráculo de padding por sí solo para descifrar *cualquier* texto cifrado de su elección. Esto es cierto sin importar qué más haga el servidor web. Siempre que filtre este único bit de información sobre textos cifrados que el atacante puede elegir, bien podría estar filtrando todo.

11.1.1. Maleabilidad del Cifrado CBC

Recuerda la definición del descifrado CBC. Si el texto cifrado es $c = c_0||c_1||\ldots||c_t$, entonces el i-ésimo bloque de texto plano se calcula como:

$$m_i := F^{-1}(k, c_i) \oplus c_{i-1}.$$

De esto podemos deducir dos hechos importantes:

- Dos bloques consecutivos (c_{i-1}, c_i) tomados aisladamente son un cifrado válido de m_i. Mirando hacia adelante, este hecho le permite al atacante enfocarse en descifrar un solo bloque a la vez.
- Hacer XOR de un bloque del texto cifrado con un valor conocido (digamos, x) tiene el efecto de hacer XOR del bloque de texto plano correspondiente con el mismo valor. En otras palabras, para todo x, el texto cifrado $(c_{i-1} \oplus x, c_i)$ se descifra como $m_i \oplus x$:

$$\text{Dec}(k, (c_{i-1} \oplus x, c_i)) = F^{-1}(k, c_i) \oplus (c_{i-1} \oplus x) = (F^{-1}(k, c_i) \oplus c_{i-1}) \oplus x = m_i \oplus x.$$

Si enviamos tal texto cifrado $(c_{i-1} \oplus x, c_i)$ al oráculo de padding, por lo tanto, aprenderíamos si $m_i \oplus x$ es un (solo bloque) con padding válido.

Al elegir cuidadosamente diferentes valores x y hacer preguntas de esta forma al oráculo de padding, mostraremos cómo es posible aprender todo m_i. Resumimos la capacidad hasta ahora con la siguiente subrutina:

[1] Por esta razón, es necesario escribir el algoritmo de eliminación del padding de manera que cada camino de ejecución a través de la subrutina tome el mismo número de ciclos de CPU.

\\ supongamos que c cifra un texto plano (desconocido) $m_1||\ldots||m_\ell$
\\ ¿$m_i \oplus x$ termina en una de las cadenas de padding válidas?

CHECKXOR(c, i, x):
return PADDINGORACLE $(c_{i-1} \oplus x, c_i)$

Dado un texto cifrado c que cifra un mensaje desconocido m, podemos ver que un adversario puede generar otro texto cifrado cuyos contenidos están relacionados con m de una manera predecible.

Esta propiedad de un esquema de cifrado se llama **maleabilidad**.

11.1.2. Aprendiendo el último byte de un bloque

Ahora mostramos cómo usar la subrutina CHECKXOR para determinar el último byte de un bloque de texto plano m.

En el mundo real, los datos casi siempre vienen en *bytes* y no en bits, por lo que esa será nuestra suposición aquí. En esta sección escribiremos los bytes en hexadecimal, por ejemplo, `8f`. Las longitudes de bloque típicas son 128 bits (16 bytes) o 256 bits (32 bytes).

Un ejemplo de esquema de padding es el estándar ANSI X.923: Los datos se rellenan con bytes nulos, excepto el último byte del pad, que indica cuántos bytes de padding hay. En esencia, el último byte del mensaje le dice al receptor cuántos bytes debe eliminar para recuperar el mensaje original.

Tenga en cuenta que en este esquema de padding (y, de hecho, en todos ellos), si los datos originales sin rellenar ya son un múltiplo de la longitud del bloque, entonces se debe agregar un bloque completo adicional de relleno. Esto es necesario porque es posible que los datos originales terminen con algunos bytes que se parezcan a un relleno válido (por ejemplo, `00 00 03`), y no queremos que estos bytes se eliminen erróneamente.

Ejemplo 11.1. A continuación, se muestran algunos ejemplos de relleno X.923 válido e inválido (utilizando bloques de 16 bytes):

```
01 34 11 d9 81 88 05 57 1d 73 c3 00 00 00 00 05 ⇒ válido
95 51 05 4a d6 5a a3 44 af b3 85 00 00 00 00 03 ⇒ válido
71 da 77 5a 5e 77 eb a8 73 c5 50 b5 81 d5 96 01 ⇒ válido
5b 1c 01 41 5d 53 86 4e e4 94 13 e8 7a 89 c4 71 ⇒ inválido
d4 0d d8 7b 53 24 c6 d1 af 5f d6 f6 00 c0 00 04 ⇒ inválido
```

Hay dos casos a considerar, dependiendo del contenido de m. El atacante inicialmente no sabe qué caso se cumple:

Para el primero (y más fácil) de los dos casos, supongamos que el penúltimo byte de m es distinto de cero. Probaremos cada byte posible b y preguntaremos si $m \oplus b$ tiene un padding válido. Dado que m es un bloque y b es un solo byte, cuando escribimos $m \oplus b$ queremos decir que b se extiende a la izquierda con ceros. Dado que el penúltimo byte de m (y por lo tanto de $m \oplus b$) es distinto de cero, solo una de estas posibilidades tendrá un padding válido: aquella en la que $m \oplus b$ termina en el byte `01`. Por lo tanto, si b es el

byte candidato que tiene éxito (es decir, $m \oplus b$ tiene un padding válido), entonces el último byte de m debe ser $b \oplus \texttt{01}$.

Ejemplo 11.2. Usando `LEARNLASTBYTE` para aprender el último byte de un bloque de texto plano:

	...	a0 42 ??	m = texto plano desconocido
$\oplus$	...	00 00 b	b = byte que causa que el oráculo produzca true
=	...	a0 42 01	padding válido $\iff b \oplus \texttt{??} = \texttt{01}$
			$\iff \texttt{??} = b \oplus \texttt{01}$

Para el otro caso, supongamos que el penúltimo byte de m es cero. Entonces $m \oplus b$ tendrá un padding válido para varios valores candidatos de b:

Ejemplo 11.3. Usando `LEARNLASTBYTE` para aprender el último byte de un bloque de texto plano:

	...	a0 00 ??		...	a0 00 ??	m = bloque de texto plano desconocido
$\oplus$	...	00 00 b_1	$\oplus$	...	00 00 b_2	b_i = bytes candidatos
=	...	a0 00 01	=	...	a0 00 02	dos candidatos hacen que el oráculo devuelva verdadero
		↓			↓	
	...	a0 00 ??		...	a0 00 ??	
$\oplus$	...	00 01 b_1	$\oplus$	...	00 01 b_2	mismos b_1, b_2, pero cambiando el penúltimo byte
=	...	a0 01 01	=	...	a0 01 02	solo uno hace que el oráculo devuelva verdadero
						$\Rightarrow ?? = b_1 \oplus 01$

Siempre que más de un valor candidato b produce un padding válido, sabemos que el penúltimo byte de m es cero (de hecho, al contar el número de candidatos exitosos, podemos saber exactamente cuántos ceros preceden al último byte de m).

Si el penúltimo byte de m es cero, entonces el penúltimo byte de $m \oplus \texttt{01}\ b$ es distinto de cero. La única forma de que ambas cadenas $m \oplus \texttt{01}\ b$ y $m \oplus b$ tengan un padding válido es cuando $m \oplus b$ termina en el byte `01`. Podemos volver a probar todos los valores candidatos exitosos de b, esta vez con un byte adicional distinto de cero al frente. Habrá un único candidato b que tenga éxito en ambas rondas, y esto nos dirá que el último byte de m es $b \oplus \texttt{01}$.

El enfoque general para aprender el último byte de un bloque de texto plano se resume en la subrutina `LEARNLASTBYTE` en la Figura 11.1. El conjunto B contiene los bytes candidatos exitosos de la primera ronda. Hay como máximo 16 elementos en B después de la primera ronda, ya que solo hay 16 posibilidades válidas para el último byte de un bloque con un padding adecuado. En el peor de los casos, `LEARNLASTBYTE` realiza $256 + 16 = 272$ llamadas al oráculo de padding (a través de CHECKXOR).

11.1.3. Aprendiendo otros bytes de un bloque

Una vez que hemos aprendido uno de los bytes finales de un bloque de texto plano, es ligeramente más fácil aprender bytes adicionales. Supongamos que conocemos los últimos 3 bytes de un bloque de texto plano,

como en el ejemplo siguiente. Nos gustaría usar el oráculo de padding para descubrir el cuarto byte desde el final.

Ejemplo 11.4. Usando `LEARNPREVBYTE` para aprender el cuarto byte desde el final cuando los últimos 3 bytes del bloque ya se conocen.

	...	`?? a0 42 3c`	m = bloque de texto plano parcialmente desconocido
$\oplus$	...	`00 00 00 04`	p = cadena que termina en `04`
$\oplus$	...	`00 a0 42 3c`	s = bytes conocidos de m
$\oplus$	...	b `00 00 00`	y = byte candidato b desplazado a su posición
$=$	...	`00 00 00 04`	padding válido $\Leftrightarrow$?? $= b$

Puesto que conocemos los últimos 3 bytes de m, podemos calcular una cadena x tal que $m \oplus x$ termine en `00 00 04`. Ahora podemos intentar hacer XOR del cuarto byte desde el final de $m \oplus x$ con diferentes bytes candidatos b, y preguntar al oráculo de padding si el resultado tiene un padding válido. Un padding válido solo ocurre cuando el resultado tiene `00` en su cuarto byte desde el final, y esto ocurre exactamente cuando el cuarto byte desde el final de m coincide exactamente con nuestro byte candidato b.

El proceso se resume en la subrutina `LEARNPREVBYTE` en la Figura 11.1. En el peor de los casos, esta subrutina realiza 256 consultas al oráculo de padding.

Juntando todo. Ahora tenemos todas las herramientas necesarias para descifrar cualquier texto cifrado usando solo el oráculo de padding. El proceso se resume a continuación en la subrutina `LEARNALL`.

En el peor de los casos, se requieren 256 consultas al oráculo de padding para aprender cada byte del texto plano.[2] Sin embargo, en la práctica, el número puede ser mucho menor. El ejemplo de esta sección se inspiró en un ataque real de oráculo de padding [3] que incluye optimizaciones que permiten a un atacante recuperar cada byte del texto plano con solo 14 consultas al oráculo en promedio.

11.2. ¿Qué salió mal?

El cifrado CBC proporciona seguridad CPA, entonces, ¿por qué no nos salvó de los ataques de oráculo de padding? ¿Cómo fue capaz un atacante de aniquilar por completo la privacidad del cifrado?

1. Primero, el cifrado CBC (de hecho, cada esquema de cifrado que hemos visto hasta ahora) tiene una propiedad llamada **maleabilidad**. Dado un cifrado c de un texto plano *desconocido* m, es posible generar otro texto cifrado c' cuyos contenidos están *relacionados con m de una manera predecible*. En el caso del cifrado CBC, si el texto cifrado $c_0||\cdots||c_\ell$ encripta un texto plano $m_1||\cdots||m_\ell$, entonces el texto cifrado $(c_{i-1} \oplus x, c_i)$ encripta el texto plano relacionado $m_i \oplus x$.

   En resumen, si un esquema de cifrado es maleable, entonces permite que la información contenida en un texto cifrado se "transfiera" a otro texto cifrado.

[2] Podría tomar más de 256 consultas para aprender el último byte. Pero cada vez que LEARNLASTBYTE utiliza más de 256 consultas, el efecto secundario es que también has descubierto que algunos otros bytes del bloque son cero. Esto te evita tener que consultar el oráculo de padding por completo para aprender esos bytes.

[3] *How to Break XML Encryption*, Tibor Jager and Juraj Somorovsky. ACM CCS 2011.

CHECKXOR(c, i, x):
// Si c encripta un texto plano
// (desconocido) $m_1||\cdots||m_\ell$; entonces
// ¿$m_i \oplus x$ (por sí mismo)
// tiene un padding válido?
return PADDINGORACLE$(c_{i-1} \oplus x, c_i)$

LEARNLASTBYTE(c, i):
// deduce el último byte del
// bloque de texto plano m_i
$B := \emptyset$
for $b =$ `00` to `ff`:
  if CHECKXOR(c, i, b):
    $B := B \cup \{b\}$
if $|B| = 1$:
  $b :=$ único elemento de B
  return $b \oplus$ `01`
else:
  for each $b \in B$:
    if CHECKXOR$(c, i,$ `01` $b)$:
      return $b \oplus$ `01`

LEARNALL(c):
// aprende todo el texto plano
$m_1||\cdots||m_\ell$
$m := \varepsilon$
$\ell :=$ número de bloques no IV en c
for $i = 1$ to ℓ:
  $m := m||$LEARNBLOCK(c, i)
return m

LEARNPREVBYTE(c, i, s):
// sabiendo que m_i termina en s,
// encuentra el byte más a la derecha
// desconocido de m_i
$p := |s| + 1$
for $b =$ `00` to `ff`:
  $y := b\underbrace{\texttt{00 ... 00}}_{|s|}$
  if CHECKXOR$(c, i, p \oplus s \oplus y)$:
    return b

LEARNBLOCK(c, i):
// aprende el bloque de texto plano
// completo m_i
$s :=$LEARNLASTBYTE(c, i)
do 15 veces:
  $b :=$ LEARNPREVBYTE(c, i, s)
  $s := b||s$
return s

Figura 11.1: Resumen del ataque con el oráculo de padding.

2. Segundo, puede que hayas notado que la definición de seguridad CPA no menciona el algoritmo Dec. El algoritmo Dec aparece en nuestra definición de corrección, pero no se encuentra en las librerías $\mathcal{L}_{\text{cpa-0}}$ y $\mathcal{L}_{\text{cpa-1}}$. ¡El descifrado no tiene impacto en la seguridad CPA!

   Pero el contexto del oráculo de padding involucraba al algoritmo Dec; en particular, al adversario se le permitía ver cierta información sobre el resultado de Dec aplicado a textos cifrados elegidos de forma adversaria. Debido a eso, la definición de seguridad CPA no captura el escenario del ataque de oráculo de padding.

La conclusión es: dale a un atacante un esquema de cifrado maleable y acceso a cualquier información parcial relacionada con el descifrado, y él/ella puede obtener información que se filtre de maneras sorprendentes. Como demuestra el ataque de oráculo de padding, incluso si solo se filtra un único bit de información (es decir, la respuesta a una pregunta de sí/no sobre un texto plano) sobre el resultado del descifrado, esto es frecuentemente suficiente para extraer todo el texto plano.

Si queremos seguridad incluso bajo el escenario del oráculo de padding, necesitamos una mejor definición de seguridad y esquemas de cifrado que la logren. De eso trata el resto de este capítulo.

Discusión

- **¿Es esta una preocupación realista?** Uno podría preguntarse si toda esta situación es algo artificial, solo para dar a los criptógrafos problemas más difíciles de resolver. Probablemente esa era una actitud común hacia las definiciones de seguridad introducidas en este capítulo. Sin embargo, en 1998, Daniel Bleichenbacher demostró un ataque devastador contra las primeras versiones del protocolo SSL. Al presentar millones de textos cifrados cuidadosamente elaborados a un servidor web, un atacante podría eventualmente recuperar claves de sesión SSL arbitrarias.

  En la práctica, es difícil hacer que el comportamiento externo de un servidor no dependa del resultado del descifrado. Esto hace que la seguridad CPA sea bastante frágil en el mundo real. En el caso de los ataques de oráculo de padding, los errores en la implementación pueden dar lugar a diferentes mensajes de error para un padding no válido. En otros casos, incluso una implementación por lo demás cuidadosa puede proporcionar un oráculo de padding a través de un canal lateral de tiempo (si el tiempo de respuesta del servidor es diferente para textos planos con padding válido/no válido).

  Como veremos, *es* de hecho posible proporcionar seguridad en este tipo de configuraciones, y con poca sobrecarga adicional. Hoy en día, rara vez hay una buena excusa para usar un cifrado que solo sea seguro bajo CPA.

- El padding está en el nombre del ataque. Pero el padding no es el culpable. El culpable es usar un esquema de cifrado (meramente) seguro bajo CPA al tiempo que se permite que se filtre cierta información sobre el resultado del descifrado.

- **Si el padding se añade solo al último bloque del texto plano, ¿cómo puede este ataque recuperar *todo* el texto plano?** Esta confusión común es otra razón para no culpar tanto al esquema de padding. Un oráculo de padding tiene el siguiente comportamiento: "dame un cifrado de $m_1||\ldots||m_\ell$ y te diré alguna información sobre m_ℓ (si termina con un cierto sufijo)". De hecho, el oráculo de padding solo revisa el último bloque. Sin embargo, el modo CBC tiene la propiedad de que si tienes un cifrado de $m_1||\ldots||m_\ell$, entonces puedes construir fácilmente un texto cifrado diferente que encripta $m_1||\ldots||m_{\ell-1}$. Si envías este texto cifrado al oráculo de padding, obtendrás información

sobre $m_{\ell-1}$. Al modificar el texto cifrado (a través de la maleabilidad de CBC), le das a diferentes bloques de texto plano la oportunidad de ser el "último bloque" que el oráculo de padding examina.

- El ataque parece superficialmente como fuerza bruta, pero no lo es: El ataque realiza 256 consultas por byte de texto plano, por lo que cuesta alrededor de 256ℓ consultas para un texto plano de ℓ bytes. Forzar brutalmente todo el texto plano costaría 256^{ℓ}, ya que esa es la cantidad de textos planos de ℓ bytes que hay. Por lo tanto, el ataque es exponencialmente mejor que la fuerza bruta. La lección es: forzar brutalmente pequeñas partes a la vez es mucho mejor que forzar brutalmente todo.

11.2.1. Maleabilidad CPA

Ya que hemos mencionado la *maleabilidad* anteriormente, dedicaremos esta sección a definir la **maleabilidad CPA (frente a texto planos escogidos)**.

Recuerda que la definición informal de maleabilidad: Dado un cifrado c de un texto plano *desconocido* m, es posible generar otro texto cifrado c' cuyos contenidos están *relacionados con* m *de una manera predecible*. Esta definición era suficiente para describir el ataque usando oráculo de padding. Sin embargo, esta definición es demasiado *débil* para utilizarla en escenarios donde el adversario tiene acceso a ciertos recursos. Por ejemplo, un adverdario podría tener acceso a la librería CTXT asociada a un esquema de cifrado. Podemos fortalecer la definición de seguridad dándole más recursos al adversario en el sentido de que el cifrado c corresponda a un mensaje *conocido* por el adversario. Es más, podemos dejar que el adversario elija un cierto número de mensajes $m_1, \ldots, m_\ell$ y utilizar sus cifrados para construir c' cuyos contenidos están *relacionados con* $m_1, \ldots, m_\ell$ *de una manera predecible*.

Desafortunadamente, es complicado formalizar la definición de un esquema maleable en términos de librerías indistinguibles. Sin embargo, necesitaremos la siguiente librería:

$\mathcal{L}^{\Sigma}_{\mathrm{mCPA}}$

$k := \Sigma.\mathrm{Gen}$
$B := \emptyset$

$\underline{\mathrm{MCTXT}(\ell; m_1, \ldots, m_\ell \in \Sigma.\mathcal{M}):}$
for $i = 1$ to ℓ
  $B := B \cup \{m_i\}$
  $c_i := \Sigma.\mathrm{Enc}(k, m_i)$
return $c_1, \ldots, c_\ell$

$\underline{\mathrm{VERIFY}(c', \mathcal{R}):}$
$m' := \Sigma.\mathrm{Dec}(k, c')$
If $m' \notin \Sigma.\mathcal{M}$
  return error
return $\mathcal{R}(B, m') \stackrel{?}{=} 1$

La subrutina MCTXT guarda los mensajes $m_1, \ldots, m_\ell$ en B y los cifra todos utilizando la misma clave k.

En cuanto a la subrutina VERIFY, esta pasa dos entradas. La entrada $\mathcal{R}$ es una función de $\ell+1$ variables con rango en $\{0,1\}$. La entrada c' es un 'texto cifrado ". En la definición de seguridad mCPA, la función $\mathcal{R}$ debe describir la relación entre los contenidos de c' con los mensajes $m_1, \ldots, m_\ell$ en B. El hecho que esta función exista implica que la relación entre c' y los mensajes es predecible *sin que el adversario conozca la clave ni el texto plano m' correspondiente a la clave.* Esta subrutina verifica que esta función sea capaz de producir el valor 1 cuando se evalúa en m' y B.

Definición 11.2.1 (Propiedad mCPA)**.** *Sea Σ un esquema de cifrado. Decimos que Σ es **maleable frente a ataques con texto plano elegido (propiedad mCPA)** si existe un adversario $\mathcal{A}$ tal que la cantidad $\Pr[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{mCPA} \Rightarrow \texttt{true}]$ no es despreciable.*

Es importante remarcar que esta es no es una definición de seguridad. La maleabilidad es una propiedad un esquema de cifrado. Luego demostraremos que si un esquema tiene esta propiedad, entonces podremos deducir algo sobre su seguridad.

Desafortunadamente, es mucho más fácil demostrar que un esquema es maleable que demstrar que no lo es. Ten en cuenta que para demostrar que un esquema Σ no es maleable debes demostrar que *para todo* adversario $\mathcal{A}$, la cantidad $\Pr[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\mathrm{mCPA}} \Rightarrow \texttt{true}]$ es despreciable. La estrategia más común para este tipo de demostraciones es por contradicción, es decir, asumir que el esquema es maleable y encontrar alguna contradicción. La parte complicada es encontrar tal contradicción. Más adelante veremos que esta contradicción suele estar relacionada con problemas matemáticos que se presume que son difíciles de resolver para un adversario polinomial.

Por ahora, nos conformaremos con ver cómo utilizar la definición para demostrar que un esquema es maleable. Para ello, el punto de partida más adecuado suele ser considerar el algoritmo de descifrado del esquema y los cifrados $c_1, \ldots, c_\ell$ obtenidos al llamar MCTXT. Las operaciones realizdas por Dec con la clave k indican las modificaciones admisibles que podemos hacer a los cifrados $c_1, \ldots, c_\ell$ para obtener un texto cifrado c' asociado a un mensaje legítimo m'. La función $\mathcal{R}$ suele aparecer de manera natural tras hacer modificaciones a los textos cifrados. Ilustramos esta estrategia con el one-time pad.

Afirmación 11.2.2. *El one-time pad es maleable (mCPA).*

Demostración. Debemos construir una adversario capaz de transferir información contenida en un texto cifrado con una clave a otro texto cifrado asociado a la misma clave. Esto debe estar formalizado utilizando la librería $\mathcal{L}^{\mathrm{OTP}}_{\mathrm{mCPA}}$.

En el caso del one-time pad, fijada una clave k y un texto cifrado c, el texto plano m se obtiene haciendo el XOR entre ellos, $m := k \oplus c$. Observa que la operación XOR es asociativa, es decir, dada tres cadenas de bits del mismo tamaño u, v, w, tenemos $(u \oplus v) \oplus w = u \oplus (v \oplus w)$. Esto implica que si hacemos el XOR a ambos lados de la ecuación $m := k \oplus c$ con una cadena de bits w (del mismo tamaño que c), obtenemos

$$m \oplus w := (k \oplus c) \oplus w = k \oplus (c \oplus w).$$

Claramente, hemos obtenido un texto cifrado $c' := c \oplus w$ con la propiedad de que si desciframos con la clave k, obtenemos un mensaje $m' := m \oplus w$. Más aún, este mensaje está relacionado de manera predecible con el mensaje original m. Esta idea se formaliza en la demostración.

Consideramos el siguiente adversario:

$\mathcal{A}$

$m \leftarrow \{0,1\}^\lambda$

$c := \text{MCTXT}(1, m)$

$w \leftarrow \{0,1\}^\lambda$
$\underline{\mathcal{R}(a,b):}$
If $a = b \oplus w$ then return 1
else return 0

return VERIFY$(c \oplus w, \mathcal{R})$

Es obligatorio calcular la probabilidad de que el siguiente programa produzca `true` como salida:

$\mathcal{A}$

$m \leftarrow \{0,1\}^\lambda$

$c := \text{MCTXT}(1, m)$

$w \leftarrow \{0,1\}^\lambda$
$\underline{\mathcal{R}(a,b):}$
If $a = b \oplus w$ then return 1
else return 0

return VERIFY$(c \oplus w, \mathcal{R})$

$\diamond$

$\mathcal{L}^{\text{OTP}}_{\text{mCPA}}$

$k \leftarrow \{0,1\}^\lambda$
$B := \emptyset$

$\underline{\text{MCTXT}(\ell; m_1, \ldots, m_\ell):}$
for $i = 1$ to ℓ
  $B := B \cup \{m_i\}$
  $c_i := k \oplus m_i$
return $c_1, \ldots, c_\ell$

$\underline{\text{VERIFY}(c', \mathcal{R}):}$
$m' := k \oplus c'$
If $m' \notin \{0,1\}^\lambda$
  return error
return $\mathcal{R}(B, m') \stackrel{?}{=} 1$

Hemos incluido los detalles del OTP en $\mathcal{L}^{\text{OTP}}_{\text{mCPA}}$. Cuando $\mathcal{A}$ está vinculado a $\mathcal{L}^{\text{OTP}}_{\text{mCPA}}$, VERIFY calcula el texto plano $m' := k \oplus (c \oplus w) = (k \oplus c) \oplus w = m \oplus w$. Observa que $m \oplus w$ es una cadena de bits de longitud λ, así que m' pertenece al espacio de mensajes planos del OTP (OTP.$\mathcal{M} = \{0,1\}^\lambda$), lo cual asegura que no se producirá un error. Además, se cumple que $m' = m \oplus w$, así que $\mathcal{R}(m, m') = 1$ siempre, es decir, VERIFY$(c \oplus w, \mathcal{R})$=`true` siempre. Consecuentemente, $\Pr[\mathcal{A} \diamond \mathcal{L}^{\text{OTP}}_{\text{mCPA}} \Rightarrow \texttt{true}] = 1$, una cantidad no despreciable.

La existencia de $\mathcal{A}$ demuestra que el OTP es un esquema maleable CPA. ■

11.3. Definición de seguridad CCA

Nuestro objetivo ahora es desarrollar una nueva definición de seguridad: una que considere a un adversario que puede construir textos cifrados maliciosos y observar los efectos causados por su descifrado. Comenzaremos con el enfoque básico de la seguridad CPA, con librerías que difieren solo en cuál de dos textos planos encriptan.

En un sistema típico, un adversario podría ser capaz de aprender solo alguna información parcial específica sobre el algoritmo Dec. En el ataque de oráculo de padding, el adversario pudo aprender solo si el resultado del descifrado tenía un padding válido.

Sin embargo, estamos tratando de idear una definición de seguridad que sea útil sin importar cómo se implemente el esquema de cifrado. ¿Cómo podemos anticipar cada tipo de información parcial que podría llegar al adversario en cada uso posible del esquema de cifrado? La opción más segura es ser lo más pesimistas posible, siempre y cuando terminemos con una noción de seguridad que realmente podamos lograr al final. Así que simplemente **permitamos que el adversario descifre totalmente textos cifrados arbitrarios de su elección**. En otras palabras, si podemos garantizar la seguridad cuando el adversario tiene información completa sobre los textos cifrados desencriptados, entonces ciertamente tenemos seguridad cuando el adversario aprende solo información parcial sobre los textos cifrados desencriptados (como en un sistema típico del mundo real).

Pero esto presenta un problema significativo. Un adversario puede hacer $c^* := \text{EAVESDROP}(m_0, m_1)$ para obtener un texto cifrado de desafío, y luego pedir inmediatamente que se descifre ese texto cifrado c^*. Esto obviamente revelará al adversario a cuál librería está vinculado.

Por lo tanto, simplemente proporcionar acceso irrestricto a Dec al adversario no puede conducir a una definición de seguridad razonable (es una definición de seguridad que nunca se puede satisfacer). La forma más sencilla de solucionar este problema obvio con la definición es permitir que el adversario solicite el descifrado de cualquier texto cifrado, **excepto** los producidos en respuesta a las consultas de **EAVESDROP**. Al hacerlo, llegamos a la definición de seguridad final: seguridad contra ataques de texto cifrado elegido, o seguridad CCA:

Definición 11.3.1 (Seguridad CCA)**.** *Sea Σ un esquema de cifrado. Decimos que Σ tiene* ***seguridad contra ataques de texto cifrado elegido (seguridad CCA)*** *si $\mathcal{L}^{\Sigma}_{cca\text{-}0} \approx \mathcal{L}^{\Sigma}_{cca\text{-}1}$, donde:*

$\mathcal{L}^{\Sigma}_{cca\text{-}0}$

$k \leftarrow \Sigma.Gen$
$\mathcal{S} := \emptyset$

$\underline{EAVESDROP(m_0, m_1 \in \Sigma.\mathcal{M}):}$
If $|m_0| \neq |m_1|$ *return error*
$c := \Sigma.Enc(k, m_0)$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

$\underline{DECRYPT(c \in \Sigma.\mathcal{C}):}$
If $c \in \mathcal{S}$ *return error*
return $\Sigma.Dec(k, c)$

$\mathcal{L}^{\Sigma}_{cca\text{-}1}$

$k \leftarrow \Sigma.Gen$
$\mathcal{S} := \emptyset$

$\underline{EAVESDROP(m_0, m_1 \in \Sigma.\mathcal{M}):}$
If $|m_0| \neq |m_1|$ *return error*
$c := \Sigma.Enc(k, m_1)$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

$\underline{DECRYPT(c \in \Sigma.\mathcal{C}):}$
If $c \in \mathcal{S}$ *return error*
return $\Sigma.Dec(k, c)$

En esta definición, el conjunto $\mathcal{S}$ realiza un seguimiento de los textos cifrados que han sido generados por la subrutina EAVESDROP. La subrutina DECRYPT se niega a descifrar estos textos cifrados, pero con gusto descifrará cualquier otro texto cifrado de la elección del adversario. Esta restricción es crucial para la seguridad CCA, ya que evita que el adversario simplemente descifre el texto cifrado de desafío para ganar trivialmente el juego.

11.3.1. Un ejemplo

El ataque de oráculo de padding ya demuestra que el modo CBC no proporciona seguridad en presencia de ataques de texto cifrado elegido. Pero ese ataque fue bastante complicado, ya que el adversario estaba restringido a aprender solo 1 bit de información a la vez sobre un texto cifrado descifrado. Un ataque contra la seguridad CCA completa puede ser mucho más directo, ya que el adversario tiene acceso completo a los textos cifrados desencriptados.

Ejemplo 11.5. Considere el siguiente adversario atacando la seguridad CCA del modo CBC (con longitud de bloque *blen*):

$\mathcal{A}$

$c = c_0||c_1||c_2 := \text{EAVESDROP}(0^{2blen}, 1^{2blen})$
$m := \text{DECRYPT}(c_0||c_1)$
return $m \stackrel{?}{=} 0^{blen}$

Se puede verificar fácilmente que este adversario logra una ventaja de 1 al distinguir $\mathcal{L}_{\text{cca-0}}$ de $\mathcal{L}_{\text{cca-1}}$. El ataque utiliza el hecho (también utilizado en el ataque de oráculo de padding) de que si $c_0||c_1||c_2$ encripta $m_1||m_2$, entonces $c_0||c_1$ encripta m_1. Para nosotros, es obvio que el texto cifrado $c_0||c_1$ está relacionado con $c_0||c_1||c_2$. Desafortunadamente para el modo CBC, la definición de seguridad no es muy inteligente: dado que $c_0||c_1$ es simplemente diferente de $c_0||c_1||c_2$, la subrutina DECRYPT lo descifra felizmente.

Ejemplo 11.6. Como era de esperar, existen muchas formas muy sencillas de atacar catastróficamente la seguridad CCA del cifrado en modo CBC. Aquí hay algunos más (donde $\overline{x}$ denota el resultado de invertir cada bit en x):

$\mathcal{A}'$

$c_0||c_1||c_2 := \text{EAVESDROP}(0^{2blen}, 1^{2blen})$
$m := \text{DECRYPT}(c_0||c_1||\overline{c_2})$
if m comienza con 0^{blen} return 1 else return 0

$\mathcal{A}''$

$c_0||c_1||c_2 := \text{EAVESDROP}(0^{2blen}, 1^{2blen})$
$m := \text{DECRYPT}(\overline{c_0}||c_1||c_2)$
return $m \stackrel{?}{=} 0^{blen}||1^{blen}$

El primer ataque utiliza el hecho de que modificar c_2 no tiene ningún efecto en el primer bloque del texto plano. El segundo ataque utiliza el hecho de que invertir cada bit en el IV invierte cada bit en m_1.

Nuevamente, en todos estos casos, se está llamando a la subrutina DECRYPT con un texto cifrado diferente (pero relacionado) al que devolvió EAVESDROP.

11.3.2. Discusión

Entonces, si uso un esquema de cifrado seguro bajo CCA, ¿nunca debería descifrar un texto cifrado que yo mismo encripté?

Recuerda: cuando definimos los algoritmos Enc y Dec de un esquema de cifrado, estamos describiendo las cosas desde la perspectiva del usuario normal. Como usuario de un esquema de cifrado, puedes encriptar y descifrar lo que quieras. De hecho, sería extraño que encriptaras algo sabiendo que nunca debería ser descifrado. ¿Cuál sería el punto?

La definición de seguridad describe las cosas desde la perspectiva del atacante. Las bibliotecas $\mathcal{L}_{\text{cca-0}}$ y $\mathcal{L}_{\text{cca-1}}$ nos dicen ¿cuáles son las circunstancias bajo las cuales el esquema de cifrado proporciona seguridad? Dicen (aproximadamente):

> un atacante no puede saber qué hay dentro de un texto cifrado c^*, incluso si tiene información parcial sobre ese texto plano, incluso si tuvo alguna influencia parcial sobre la elección de ese texto plano, e incluso si se le permite descifrar cualquier otro texto cifrado que desee.

Por supuesto, si un sistema del mundo real permite que un atacante aprenda el resultado de descifrar c^*, entonces, por definición, el atacante aprende lo que hay dentro de ese texto cifrado.

La seguridad CCA está profundamente conectada con el concepto de *maleabilidad*. Maleabilidad significa que, dado un texto cifrado que encripta un texto plano desconocido m, es posible generar un texto cifrado diferente que encripta un texto plano que está relacionado con m de una manera predecible. Por ejemplo:

- Si $c_0||c_1||c_2$ es un cifrado CBC de $m_1||m_2$, entonces $c_0||c_1$ es un cifrado CBC de m_1.
- Si $c_0||c_1||c_2$ es un cifrado CBC de $m_1||m_2$, entonces $c_0||c_1||0^{blen}$ es un cifrado CBC de algún texto plano que comienza con $m_1||m_2$.
- Si $c_0||c_1$ es un cifrado CBC de m_1, entonces $(c_0 \oplus x)||c_1$ es un cifrado CBC de $m_1 \oplus x$.

Observa en el segundo ejemplo que no necesitamos conocer exactamente la relación entre los textos cifrados antiguo y nuevo.

Si un esquema de cifrado es maleable, entonces un ataque típico contra su seguridad CCA funcionaría de la siguiente manera:

1. Solicitar un cifrado c de algún texto plano.
2. Aplicando la maleabilidad del esquema, modificar c a algún otro texto cifrado c'.
3. Pedir que se descifre c'.

Dado que $c' \neq c$, la librería de seguridad permite que se descifre c'. La maleabilidad del esquema dice que el contenido de c' debe estar relacionado con el contenido de c. En otras palabras, ver el contenido de c' debería permitir que el atacante determine qué se encriptó inicialmente en c. Esta idea se puede utilizar para demostrar que el one-time pad no tiene seguridad CCA. Además, la demostración de que el one-time pad no es CCA seguro captura y simplifica los detalles del ataque con el oráculo de padding.

Afirmación 11.3.2. *El one-time pad no tiene seguridad CCA.*

Demostración. Debemos montar un ataque exitoso a la seguridad CCA del one-time pad. Ya que el one-time pad es maleable, utilizaremos la idea descrita hace dos párrafos.

Debemos construir un atacante capaz de distinguir la siguientes librerías:

$\mathcal{L}^{\text{OTP}}_{\text{cca-0}}$

$k \leftarrow \{0,1\}^\lambda$
$\mathcal{S} := \emptyset$

EAVESDROP$(m_0, m_1 \in \{0,1\}^\lambda)$:
If $|m_0| \neq |m_1|$ return error
$c := k \oplus m_0$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

DECRYPT$(c \in \{0,1\}^\lambda)$:
If $c \in \mathcal{S}$ return error
return $k \oplus c$

$\mathcal{L}^{\text{OTP}}_{\text{cca-1}}$

$k \leftarrow \{0,1\}^\lambda$
$\mathcal{S} := \emptyset$

EAVESDROP$(m_0, m_1 \in \{0,1\}^\lambda)$:
If $|m_0| \neq |m_1|$ return error
$c := k \oplus m_1$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

DECRYPT$(c \in \{0,1\}^\lambda)$:
If $c \in \mathcal{S}$ return error
return $k \oplus c$

Considere el siguiente distinguidor:

$\mathcal{A}$

$c := \text{EAVESDROP}(0^\lambda, 1^\lambda)$
$w \leftarrow \{0,1\}^\lambda$
$m := \text{DECRYPT}(c \oplus w)$
return $m \stackrel{?}{=} w$

Es obligatorio demostrar que el distinguidor se comporta de manera diferente cuando está vinculado a librerías distintas.

$\mathcal{A}$

$c := \text{EAVESDROP}(0^\lambda, 1^\lambda)$
$w \leftarrow \{0,1\}^\lambda$
$m := \text{DECRYPT}(c \oplus w)$
return $m \stackrel{?}{=} w$

$\diamond$

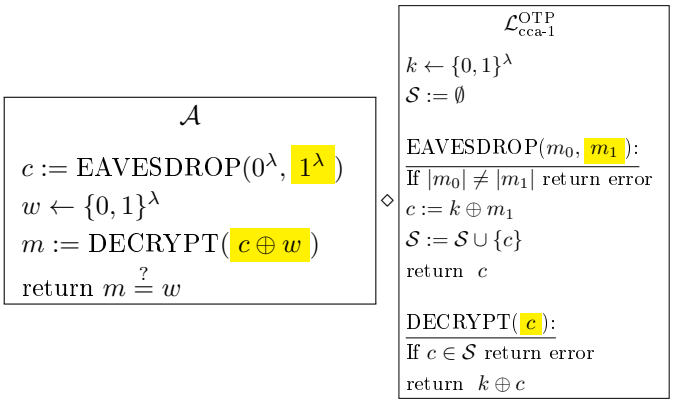
$\mathcal{L}^{\text{OTP}}_{\text{cca-0}}$

$k \leftarrow \{0,1\}^\lambda$
$\mathcal{S} := \emptyset$

EAVESDROP(m_0, m_1):
If $|m_0| \neq |m_1|$ return error
$c := k \oplus m_0$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

DECRYPT(c):
If $c \in \mathcal{S}$ return error
return $k \oplus c$

Cuando $\mathcal{A}$ está vinculado a $\mathcal{L}^{\text{OTP}}_{\text{cca-0}}$, c se calcula como $k \oplus 0^\lambda = k$ y m como $k \oplus (k \oplus w) = w$. Por lo tanto, $\mathcal{A}$ siempre devolverá `true`. En otras palabras, $\Pr[\mathcal{A} \diamond \mathcal{L}^{\text{OTP}}_{\text{cca-0}} \Rightarrow \texttt{true}] = 1$.



Cuando $\mathcal{A}$ está vinculado a $\mathcal{L}^{\text{OTP}}_{\text{cca-1}}$, c se calcula como $k \oplus 1^\lambda$ y m como $k \oplus (k \oplus 1^\lambda \oplus w) = 1^\lambda \oplus w$. Es decir, m consiste en todos los bits de w cambiados. Por lo tanto, $\mathcal{A}$ nunca devolverá `true`. En otras palabras, $\Pr[\mathcal{A} \diamond \mathcal{L}^{\text{OTP}}_{\text{cca-0}} \Rightarrow \texttt{true}] = 0$.

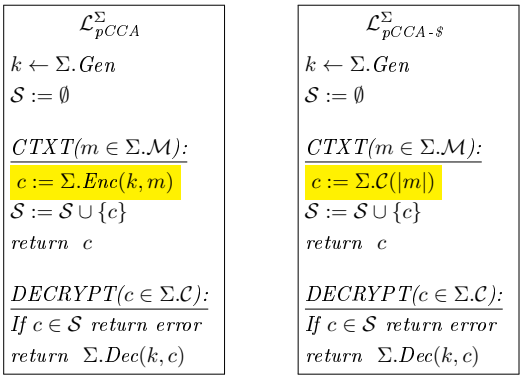
Dado que estas dos probabilidades son diferentes, esto muestra que $\mathcal{L}^{\text{OTP}}_{\text{cca-0}} \not\approx \mathcal{L}^{\text{OTP}}_{\text{cca-1}}$. En otras palabras, el one-time pad no tiene seguridad CCA.

■

11.3.3. Textos cifrados pseudoaleatorios

También podemos modificar la definición de seguridad de textos cifrados pseudoaleatorios (seguridad pCPA) de una manera similar:

Definición 11.3.3 (Seguridad pCCA)**.** *Sea Σ un esquema de cifrado. Decimos que Σ tiene* ***textos cifrados pseudoaleatorios en presencia de ataques de texto cifrado elegido (seguridad pCCA)*** *si $\mathcal{L}^{\Sigma}_{pCCA} \approx \mathcal{L}^{\Sigma}_{pCCA\text{-}\$}$ donde:*



Al igual que para la seguridad CPA, si un esquema tiene seguridad pCCA, entonces también tiene seguridad CCA, pero no viceversa.

Capítulo 12

Códigos de autenticación de mensajes

El desafío del cifrado seguro bajo CCA es lidiar con textos cifrados que fueron generados por un adversario. Imagina que hubiera una forma de "certificar" que un texto cifrado no fue generado por un adversario, es decir, que fue generado por alguien que conoce la clave secreta. Podríamos incluir dicha certificación en el texto cifrado, y el algoritmo Dec podría generar un error si se le pide que descifre algo con una certificación inválida.

Lo que estamos pidiendo no es **ocultar** el texto cifrado, sino **autenticarlo**: asegurar que fue generado por alguien que conoce la clave secreta. La herramienta para este trabajo se llama un **código de autenticación de mensajes** (MAC). Una de las aplicaciones más importantes de un código de autenticación de mensajes es transformar un esquema de cifrado seguro bajo CPA en uno seguro bajo CCA.

A medida que leas este capítulo, ten en cuenta que la privacidad y la autenticación son, de hecho, propiedades diferentes. Es posible tener una u otra o, de hecho, ambas simultáneamente. Pero una no implica la otra, y es crucial pensar en ellas por separado.

12.1. Definición

Un MAC es como una firma que se puede agregar a una parte de los datos, lo que certifica que alguien que conoce la clave secreta da fe de estos datos en particular. En criptografía, el término "firma" significa algo específico y ligeramente diferente a un MAC. En lugar de llamar a la salida de un algoritmo MAC una firma, la llamamos una "etiqueta" (o, confusamente, simplemente "un MAC").

Nuestro requisito de seguridad para un esquema MAC es que solo alguien con la clave secreta puede generar una etiqueta válida. Para verificar si una etiqueta es válida, simplemente se vuelve a calcular la etiqueta para un mensaje dado y se ve si coincide con la etiqueta reclamada. Esto implica que tanto la generación como la verificación de una etiqueta MAC requieren la clave secreta.

Definición 12.1.1 (Esquema MAC)**.** *Un esquema de código de autenticación de mensajes (MAC) para el espacio de mensajes $\mathcal{M}$ consta de los siguientes algoritmos:*

- Gen: *Muestrea una clave.*
- MAC: *Toma una clave k y un mensaje $m \in \mathcal{M}$ como entrada, y genera una* ***etiqueta*** *t. El algoritmo MAC es determinista.*

12.1.1. Cómo pensar sobre las propiedades de autenticidad

Cada definición de seguridad que hemos visto hasta ahora trata sobre ocultar información, entonces, ¿cómo hacemos una definición formal sobre la autenticidad?

Antes de ver la definición de seguridad para los MAC, comencemos con una afirmación mucho más simple (¿potencialmente obvia?): "un adversario no debería poder adivinar un valor de λ bits elegido uniformemente". Podemos formalizar esta idea con las siguientes dos librerías:

$\mathcal{L}_0$

$r \leftarrow \{0,1\}^\lambda$

GUESS(g):
return $g \stackrel{?}{=} r$

$\mathcal{L}_1$

GUESS(g):
return `false`

La librería de la izquierda permite que un programa vinculado intente adivinar una cadena "objetivo" elegida uniformemente. ¡La librería de la derecha ni siquiera se molesta en verificar la suposición del programa vinculado; de hecho, ni siquiera se molesta en muestrear una cadena objetivo aleatoria!

Las subrutinas GUESS de estas librerías dan la misma salida en casi todas las entradas. Solo hay una entrada r en la que no están de acuerdo. Si un programa logra encontrar el valor r, entonces puede distinguir fácilmente las librerías. Por lo tanto, al decir que estas librerías son indistinguibles, en realidad estamos diciendo que es **difícil para un adversario encontrar/generar este valor especial**. ¡Ese es el tipo de propiedad que queremos expresar!

De hecho, en este caso, un adversario que realiza q consultas a la subrutina GUESS logra una ventaja de como máximo $q/2^\lambda$. Para los adversarios de tiempo polinomial, esta es una ventaja insignificante (ya que q es una función polinomial de λ).

Más generalmente, supongamos que tenemos dos librerías, y una subrutina en una librería verifica alguna condición (y podría devolver verdadero o falso), mientras que en la otra librería esta subrutina siempre devuelve falso. Si las dos librerías son indistinguibles, el programa que llama no puede saber si la librería realmente está verificando la condición o si siempre está diciendo falso. Esto significa que debe ser muy difícil encontrar una entrada para la cual la respuesta "correcta" sea verdadera.

12.1.2. La definición de seguridad del MAC

Queremos decir que solo alguien que conoce la clave secreta puede crear etiquetas MAC válidas. En otras palabras, el adversario no puede crear etiquetas MAC válidas.

En realidad, esa propiedad no es suficiente para ser útil. Una propiedad más útil es: incluso si el adversario conoce etiquetas MAC válidas correspondientes a varios mensajes, no puede producir una etiqueta MAC válida para un mensaje diferente. Llamamos *falsificación* si el adversario puede producir una etiqueta MAC válida "nueva".

Para traducir esta propiedad de seguridad a una definición formal, definimos dos librerías que permiten al adversario solicitar etiquetas MAC para mensajes elegidos. Las librerías también proporcionan un mecanismo para que el adversario verifique si ha encontrado con éxito una falsificación (ya que no hay forma de verificar esta propiedad sin la clave secreta). Una librería realmente realizará la verificación, y la otra librería simplemente asumirá que las falsificaciones son imposibles. Las dos librerías solo difieren en cómo se comportan cuando el adversario llama a esta subrutina de verificación en una falsificación. Al exigir que las dos librerías sean indistinguibles, en realidad estamos exigiendo que sea difícil para el adversario generar una falsificación.

Definición 12.1.2 (Seguridad MAC)**.** *Sea Σ un esquema MAC. Decimos que Σ es un MAC seguro si $\mathcal{L}^{\Sigma}_{mac\text{-}real} \approx \mathcal{L}^{\Sigma}_{mac\text{-}fake}$ donde:*

$\mathcal{L}^{\Sigma}_{mac\text{-}real}$

$k \leftarrow \Sigma.Gen$

$\underline{GETTAG(m \in \Sigma.\mathcal{M}):}$
$return \;\; \Sigma.MAC(k, m)$

$\underline{CHECKTAG(m \in \Sigma.\mathcal{M}, t):}$
$return \;\; t \stackrel{?}{=} \Sigma.MAC(k, m)$

$\mathcal{L}^{\Sigma}_{mac\text{-}fake}$

$k \leftarrow \Sigma.Gen$
$\mathcal{T} := \emptyset$

$\underline{GETTAG(m \in \Sigma.\mathcal{M}):}$
$t := \Sigma.MAC(k, m)$
$\mathcal{T} := \mathcal{T} \cup \{(m, t)\}$
$return \;\; t$

$\underline{CHECKTAG(m \in \Sigma.\mathcal{M}, t):}$
$return \;\; (m, t) \stackrel{?}{\in} \mathcal{T}$

Discusión:

- El adversario puede ver etiquetas válidas de mensajes elegidos, a través de la subrutina GETTAG. Sin embargo, estas etiquetas no deben contar como una falsificación exitosa. La forma en que se aplica esto es en la subrutina CHECKTAG de $\mathcal{L}_{\text{mac-fake}}$; en lugar de responder siempre falso, da la respuesta correcta (verdadero) para cualquier etiqueta generada por GETTAG.
- Para que las dos librerías se comporten de manera diferente, el adversario debe llamar a CHECKTAG con la entrada (m, t) de tal manera que m nunca se usó como un argumento para GETTAG (para que $\mathcal{L}_{\text{mac-fake}}$ responda falso) pero donde la etiqueta sea realmente correcta (para que $\mathcal{L}_{\text{mac-real}}$ responda verdadero).
- El adversario puede distinguir con éxito si encuentra *cualquier* falsificación: una etiqueta MAC válida de cualquier mensaje "nuevo". A la definición no le importa si es la etiqueta de un mensaje significativo en particular.

12.1.3. Aplicaciones MAC

Aunque los MAC están menos arraigados en la conciencia pública que el cifrado, son extremadamente útiles. Una aplicación frecuente de los MAC es almacenar cierta información en un lugar no confiable, donde

no pretendemos ocultar los datos, solo asegurarnos de que los datos no se modifiquen.

- Una *cookie* de navegador es una pequeña porción de datos que un servidor web almacena en el navegador web de un usuario. El navegador presenta los datos de la cookie al servidor en cada solicitud.

  Imagina un servidor web que almacena una cookie cuando un usuario inicia sesión, que contiene el nombre de la cuenta de ese usuario. ¿Qué impide que un atacante modifique su cookie para que contenga el nombre de la cuenta de un usuario diferente? Agregar una etiqueta MAC de los datos de la cookie (usando una clave conocida solo por el servidor) asegura que tal ataque no tenga éxito. El servidor puede confiar en cualquier dato de cookie cuya etiqueta MAC sea correcta.

- Cuando Alice inicia una conexión de red con Bob, deben realizar un protocolo de enlace TCP:

  1. Alice envía un paquete SYN especial que contiene su número de secuencia inicial A. En TCP, todos los paquetes de Alice a Bob incluyen un número de secuencia, lo que ayuda a las partes a detectar cuándo faltan paquetes o están fuera de orden. Es importante que el número de secuencia inicial sea aleatorio para evitar que otras partes inyecten paquetes falsos.
  2. Bob envía un paquete SYN+ACK especial que contiene $A + 1$ (para reconocer el valor A de Alice) y el número de secuencia inicial B para sus paquetes.
  3. Alice envía un paquete ACK especial que contiene $B + 1$, y luego se establece la conexión.

  Cuando Bob está esperando el paso 3, la conexión se considera "semiabierta". Mientras espera, Bob debe recordar B para poder compararlo con el $B + 1$ que se supone que Alice enviará en su ACK final. Normalmente, el sistema operativo asigna solo una cantidad muy limitada de recursos para estas conexiones semiabiertas.

  En el pasado, era posible realizar un ataque de denegación de servicio iniciando una gran cantidad de conexiones TCP con un servidor, pero sin enviar nunca el paquete ACK final. La cola del servidor para las conexiones semiabiertas se llena, lo que impide que se inicien otras conexiones legítimas.

  Una solución ingeniosa compatible con versiones anteriores a este problema se llama **SYN cookies**. La idea es permitir que Bob elija su número de secuencia inicial B para que sea un MAC de la dirección IP del cliente, el número de puerto y otros valores. Ahora no hay nada que almacenar para las conexiones semiabiertas. Cuando Alice envía el ACK final del protocolo de enlace, Bob puede volver a calcular el número de secuencia inicial a partir de su clave MAC.

Estos son todos los casos en los que la persona que genera el MAC es la misma que luego verifica el MAC. Puedes pensar en esta persona como si eligiera no almacenar cierta información, sino dejar la información con otra persona como una "nota para sí misma".

Existen otros ajustes útiles en los que una parte genera un MAC mientras que la otra verifica.

- En la autenticación de dos factores, un usuario inicia sesión en un servicio utilizando algo que conoce (por ejemplo, una contraseña) y algo que tiene (por ejemplo, un teléfono móvil). El mecanismo de autenticación de dos factores más común se llama contraseñas únicas basadas en tiempo (TOTP). Cuando tú (como usuario) habilitas la autenticación de dos factores, generas una clave secreta k y la almacenas tanto en tu teléfono como con el proveedor del servicio. Cuando deseas iniciar sesión, abres una aplicación simple en tu teléfono que calcula $p = \text{MAC}(k, T)$, donde T es la fecha + hora actual

(generalmente redondeada a los 30 segundos más cercanos). El valor p es la "contraseña única basada en tiempo". Luego, inicias sesión en el servicio utilizando tu contraseña habitual (a largo plazo) y la contraseña única p. El proveedor del servicio tiene k y también conoce la hora actual, por lo que puede verificar el MAC p.

Desde el punto de vista del proveedor del servicio, el único otro lugar donde existe k es en el teléfono de este usuario en particular. Intuitivamente, la única forma de generar una contraseña única válida en el momento T es estar en posesión de este teléfono en el momento T. Incluso si un atacante ve tanto tu contraseña a largo plazo como tu contraseña única por encima de tu hombro, esto no le ayuda a obtener acceso a tu cuenta en el futuro (bueno, no después de 30 segundos en el futuro).

12.2. MACs para mensajes largos

Usar una PRF como MAC solo es útil para mensajes cortos de longitud fija, ya que la mayoría de las PRFs que existen en la práctica están limitadas a tales entradas. ¿Podemos de alguna manera extender una PRF para construir un esquema MAC para mensajes largos, de manera similar a como usamos los modos de cifrado en bloque para construir el cifrado para mensajes largos?

12.2.1. Cómo NO hacerlo

Para comprender los desafíos de construir un MAC para mensajes largos, primero exploramos algunos enfoques que no funcionan. Las cosas que pueden salir mal en un MAC inseguro son bastante diferentes en carácter a las cosas que pueden salir mal en un modo de cifrado en bloque, ¡así que presta mucha atención!

Ejemplo 12.1. Sea F una PRF con $in = out = \lambda$. A continuación, se muestra un enfoque MAC para mensajes de longitud 2λ. Está inspirado en el modo ECB, por lo que sabes que va a ser un desastre:

$\underline{\text{ECBMAC}(k, m_1||m_2)}$:
$t_1 := F(k, m_1)$
$t_2 := F(k, m_2)$
return $t_1||t_2$

Un problema con este enfoque es que, aunque la PRF autentica cada bloque m_1, m_2 individualmente, no hace nada para autenticar que m_1 sea el primer bloque y m_2 el segundo. Traduciendo esta observación a un ataque, un adversario puede pedir la etiqueta MAC de $m_1||m_2$ y luego predecir/falsificar la etiqueta para $m_2||m_1$:

$\mathcal{A}$
$t_1||t_2 := \text{GETTAG}(0^\lambda||1^\lambda)$
return $\text{CHECKTAG}(1^\lambda||0^\lambda,\ t_2||t_1)$

Cuando $\mathcal{A}$ está vinculado a $\mathcal{L}_{\text{mac-real}}$, siempre devuelve verdadero, ya que podemos decir que $t_2||t_1$ es de hecho la etiqueta válida para $1^\lambda||0^\lambda$. Cuando $\mathcal{A}$ está vinculado a $\mathcal{L}_{\text{mac-fake}}$, siempre devuelve falso, ya que $\mathcal{A}$ nunca llamó a GETTAG con la entrada $1^\lambda||0^\lambda$. Por lo tanto, $\mathcal{A}$ distingue las librerías con ventaja 1.

Esta ingenua construcción MAC trata a ambos m_1 y m_2 de forma idéntica, y una forma obvia de intentar solucionar el problema es tratar los diferentes bloques de forma diferente de alguna manera:

Ejemplo 12.2. Sea F una PRF con $in = \lambda + 1$ y $out = \lambda$. A continuación, se muestra otro enfoque MAC para mensajes de longitud 2λ:

$\underline{\text{ECB++MAC}(k, m_1||m_2)\text{:}}$
$t_1 := F(k, 0||m_1)$
$t_2 := F(k, 1||m_2)$
return $t_1||t_2$

Esta construcción MAC lo hace mejor, ya que trata los dos bloques de mensaje m_1 y m_2 de forma diferente. Ciertamente, el ataque anterior de intercambiar el orden de m_1 y m_2 ya no funciona. (¿Puedes ver por qué?)

La construcción autentica (en cierto sentido) el hecho de que m_1 es el primer bloque de mensaje, y m_2 es el segundo bloque. Sin embargo, esta construcción no autentica el hecho de que este m_1 y m_2 **en particular pertenecen juntos**. Más concretamente, podemos "mezclar y combinar" bloques de la etiqueta correspondientes a diferentes mensajes:

$\mathcal{A}$
$t_1||t_2 := \text{GETTAG}(0^{2\lambda})$
$t'_1||t'_2 := \text{GETTAG}(1^{2\lambda})$
return $\text{CHECKTAG}(0^\lambda||1^\lambda,\ t_1||t'_2)$

En este ataque, combinamos el bloque t_1 de la primera etiqueta y el bloque t'_2 de la segunda etiqueta.

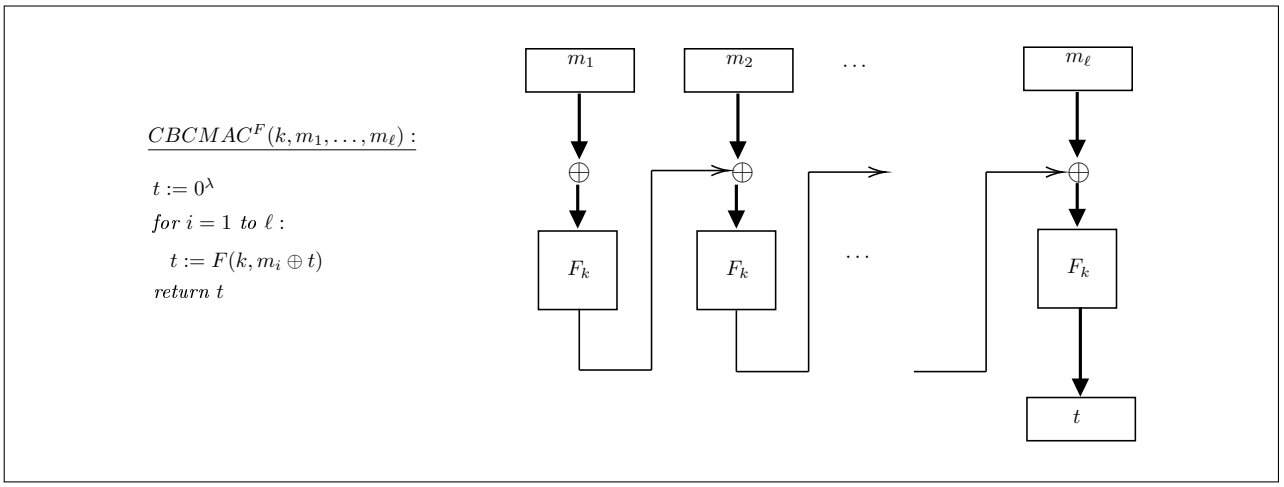
Estamos empezando a ver los desafíos que implica la construcción de un esquema MAC para mensajes largos. Un MAC seguro debe autenticar cada bloque del mensaje, el orden de los bloques del mensaje y el hecho de que *estos particulares* bloques de mensaje aparecen en un solo mensaje. En resumen, debe autenticar la *totalidad* del mensaje.

Piensa en cómo la autenticación es significativamente diferente a la privacidad/ocultamiento en este aspecto. Al menos para la seguridad CPA, podemos ocultar un texto plano completo ocultando cada parte individual del texto plano por separado (cifrándola con un cifrado seguro CPA). La autenticación es fundamentalmente diferente.

12.2.2. Cómo hacerlo: CBC-MAC

Hemos visto algunas formas inseguras de construir un MAC para mensajes más largos. Ahora veamos una forma segura. Un enfoque común para construir un MAC para mensajes largos involucra el modo de cifrado en bloque CBC.

Definición 12.2.1 (CBC-MAC). *Sea F una PRF con $in = out = \lambda$. CBC-MAC se refiere al siguiente esquema MAC:*



A diferencia del cifrado CBC, CBC-MAC no usa un vector de inicialización (o, puedes pensar que usa el IV de todos ceros), y solo emite el último bloque.

Teorema 12.2.2. *Si F es una PRF segura con $in = out = \lambda$, entonces para cualquier ℓ fijo, CBC-MAC es un MAC seguro cuando se usa con el espacio de mensajes $\mathcal{M} = \{0,1\}^{\ell\lambda}$.*

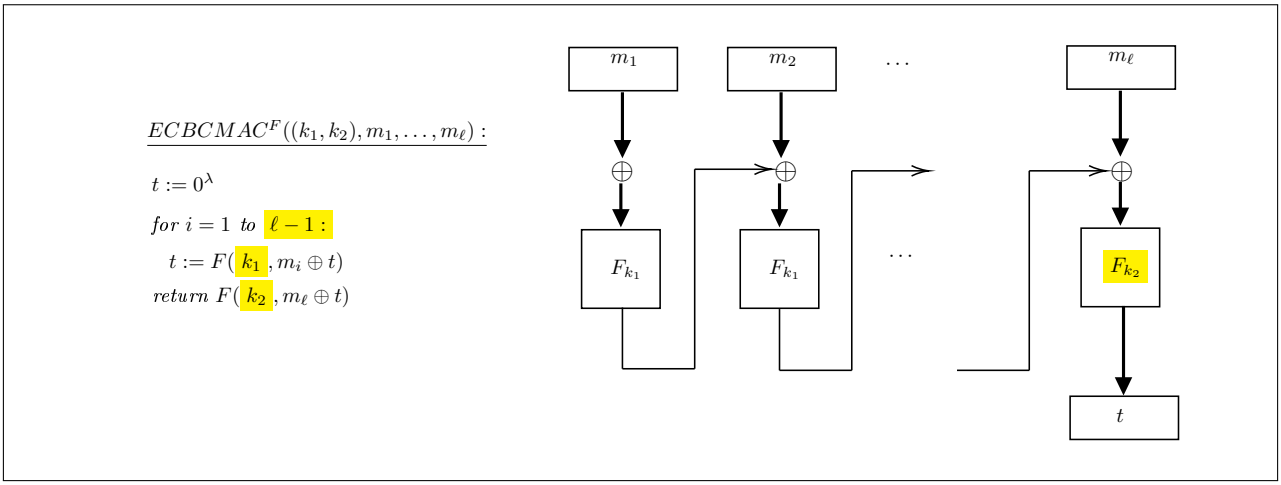
Presta mucha atención a la declaración de seguridad. Dice que si solo autenticas mensajes de 4 bloques, CBC-MAC es seguro. Si solo autenticas mensajes de 24 bloques, CBC-MAC es seguro. Sin embargo, si deseas autenticar *tanto* mensajes de 4 bloques como de 24 bloques (es decir, con la misma clave), entonces CBC-MAC no es seguro. En particular, ver el CBC-MAC de varios mensajes de 4 bloques permite a un atacante generar una falsificación de un mensaje de 24 bloques.

12.2.3. CBC-MAC más bobusto

Si CBC-MAC es tan frágil, ¿existe una manera de extenderlo para que funcione con mensajes de longitudes mixtas?

Un enfoque se llama ECBC-MAC y se muestra a continuación. Funciona tratando el último bloque de manera diferente; específicamente, utiliza una clave PRF independiente para el último bloque en la cadena CBC.

Definición 12.2.3 (ECBC-MAC). *Sea F una PRF con $in = out = \lambda$. ECBC-MAC se refiere al siguiente esquema MAC:*



Teorema 12.2.4. *Si F es una PRF segura con $in = out = \lambda$, entonces ECBC-MAC es un MAC seguro para el espacio de mensajes $\mathcal{M} = (\{0,1\}^\lambda)^*$.*

En otras palabras, ECBC-MAC es seguro para usar con mensajes de cualquier longitud (que sea un múltiplo de la longitud del bloque).

Para extender ECBC-MAC a mensajes de *cualquier* longitud (no necesariamente un múltiplo de la longitud del bloque), se puede usar un esquema de relleno como en el caso del cifrado.[1]

12.3. Cifrar-luego-MAC

Nuestra motivación para estudiar los MAC es que parecen útiles para construir un esquema de cifrado seguro bajo CCA. La idea es agregar un MAC a un esquema de cifrado seguro bajo CPA. El algoritmo de descifrado puede generar un error si el MAC no es válido, asegurando así que los textos cifrados generados por un adversario (o modificados por un adversario) no sean aceptados. Hay varias formas naturales de combinar un esquema MAC y de cifrado, *¡pero no todas son seguras!* La forma más segura se conoce como cifrar-luego-MAC:

Definición 12.3.1 (Encrypt-then-MAC (EtM))**.** *Sea E un esquema de cifrado, y M un esquema MAC donde $E.\mathcal{C} \subseteq M.\mathcal{M}$ (es decir, el esquema MAC es capaz de generar MACs de textos cifrados en el esquema E). Entonces, sea EtM la construcción* ***cifrar-luego-MAC*** *que se proporciona a continuación:*

$\mathcal{K} = E.\mathcal{K} \times M.\mathcal{K}$	$\underline{Gen:}$	$\underline{Enc((k_e,k_m),m):}$	$\underline{Dec((k_e,k_m),(c,t)):}$
$\mathcal{M} = E.\mathcal{M}$	$k_e \leftarrow E.Gen$	$c := E.Enc(k_e, m)$	$if\ t \neq M.MAC(k_m, c)\ then:$
$\mathcal{C} = E.\mathcal{C} \times M.\mathcal{T}$	$k_m \leftarrow M.Gen$	$t := M.MAC(k_m, c)$	$\quad return\ error$
	$return\ (k_e, k_m)$	$return\ (c,t)$	$return\ E.Dec(k_e, c)$

[1] Nota que si el mensaje ya es un múltiplo de la longitud del bloque, entonces el padding agrega un bloque adicional. Existen métodos ingeniosos para evitar un bloque de padding adicional en el caso de los MACs, los cuales no discutiremos aquí.

¡Es importante destacar que el esquema calcula un MAC del *texto cifrado CPA*, y no del texto plano! El resultado es un esquema de cifrado seguro bajo CCA:

Afirmación 12.3.2. *Si E tiene seguridad CPA y M es un MAC seguro, entonces EtM tiene seguridad CCA.*

Demostración. Como de costumbre, probamos la afirmación con una secuencia de librerías:

$\mathcal{L}^{\text{EtM}}_{\text{cca-0}}$

$k_e \leftarrow E.\text{Gen}$
$k_m \leftarrow M.\text{Gen}$
$\mathcal{S} := \emptyset$

EAVESDROP(m_0, m_1):
If $|m_0| \neq |m_1|$ return error
$c := E.\text{Enc}(k_e, m_0)$
$t \leftarrow M.\text{MAC}(k_m, c)$
$\mathcal{S} := \mathcal{S} \cup \{ (c,t) \}$
return (c,t)

DECRYPT(c,t):
If $(c,t) \in \mathcal{S}$ return error
if $t \neq M.\text{MAC}(k_m, c)$ then:
return error
return $E.\text{Dec}(k_e, c)$

El punto de partida es $\mathcal{L}^{\text{EtM}}_{\text{cca-0}}$, que se muestra aquí con los detalles de la construcción cifrar-luego-MAC resaltados. Nuestro objetivo es eventualmente intercambiar m_0 con m_1. Pero la seguridad CPA de E debería permitirnos hacer precisamente eso, entonces, ¿cuál es el problema?

Para aplicar la seguridad CPA de E, debemos factorizar la llamada relevante a $E.\text{Enc}$ en términos de la biblioteca CPA $\mathcal{L}^{E}_{\text{cpa-0}}$. Esto significa que k_e se vuelve privada para la librería $\mathcal{L}^{E}_{\text{cpa-0}}$. Pero k_e también se usa en la última línea de la librería en $E.\text{Dec}(k_e, c)$. ¡La librería de seguridad CPA para E no proporciona ninguna forma de llevar a cabo tales sentencias $E.\text{Dec}$!

$k_e \leftarrow E.\text{Gen}$
$\mathcal{S} := \emptyset$

EAVESDROP(m_0, m_1):
If $|m_0| \neq |m_1|$ return error
$c := E.\text{Enc}(k_e, m_0)$
$t :=$ GETTAG(c)
$\mathcal{S} := \mathcal{S} \cup \{(c,t)\}$
return (c,t)

DECRYPT(c,t):
If not CHECKTAG(c,t) return error
if $t \neq M.\text{MAC}(k_m, c)$ then:
return error
return $E.\text{Dec}(k_e, c)$

$\diamond$

$\mathcal{L}^{M}_{\text{mac-real}}$

$k_m \leftarrow M.\text{Gen}$

GETTAG(c):
return $M.\text{MAC}(k_m, c)$

CHECKTAG(c,t):
return $t \stackrel{?}{=} M.\text{MAC}(k_m, c)$

Las operaciones del esquema MAC se han factorizado en términos de $\mathcal{L}^{M}_{\text{mac-real}}$. Notablemente, en la subrutina DEC, la condición "$t \neq M.\text{MAC}(k_m, c)$" se ha reemplazado con "no CHECKTAG(c,t)".

$k_e \leftarrow E.\text{Gen}$
$\mathcal{S} := \emptyset$

EAVESDROP(m_0, m_1):
If $|m_0| \neq |m_1|$ return error
$c := E.\text{Enc}(k_e, m_0)$
$t :=$ GETTAG(c)
$\mathcal{S} := \mathcal{S} \cup \{(c,t)\}$
return (c,t)

DECRYPT(c,t):
If not CHECKTAG(c,t) return error
if $t \neq M.\text{MAC}(k_m, c)$ then:
return error
return $E.\text{Dec}(k_e, c)$

$\diamond$

$\mathcal{L}^M_{\text{mac-fake}}$

$k_m \leftarrow M.\text{Gen}$
$\mathcal{T} := \emptyset$

GETTAG(c):
$t := M.\text{MAC}(k_m, c)$
$\mathcal{T} := \mathcal{T} \cup \{(c,t)\}$
return t

CHECKTAG(c,t):
return $(c,t) \stackrel{?}{\in} \mathcal{T}$

Hemos aplicado la seguridad del esquema MAC y hemos reemplazado $\mathcal{L}_{\text{mac-real}}$ con $\mathcal{L}_{\text{mac-fake}}$.

$k_e \leftarrow E.\text{Gen}$
$k_m \leftarrow M.\text{Gen}$
$\mathcal{T} := \emptyset$
$\mathcal{S} := \emptyset$

EAVESDROP(m_0, m_1):
If $|m_0| \neq |m_1|$ return error
$c := E.\text{Enc}(k_e, m_0)$
$t \leftarrow M.\text{MAC}(k_m, c)$
$\mathcal{T} := \mathcal{T} \cup \{(c,t)\}$
$\mathcal{S} := \mathcal{S} \cup \{(c,t)\}$
return (c,t)

DECRYPT(c,t):
If $(c,t) \in \mathcal{S}$ return error
If $(c,t) \notin \mathcal{T}$ return error
return $E.\text{Dec}(k_e, c)$

Hemos integrado la librería $\mathcal{L}_{\text{mac-fake}}$. Esta librería realiza un seguimiento de un conjunto $\mathcal{S}$ de valores para el propósito de la interfaz CCA, pero también un conjunto $\mathcal{T}$ de valores para los propósitos del MAC. Sin embargo, está claro a partir del código de esta librería que $\mathcal{S}$ y $\mathcal{T}$ siempre tienen el mismo contenido.

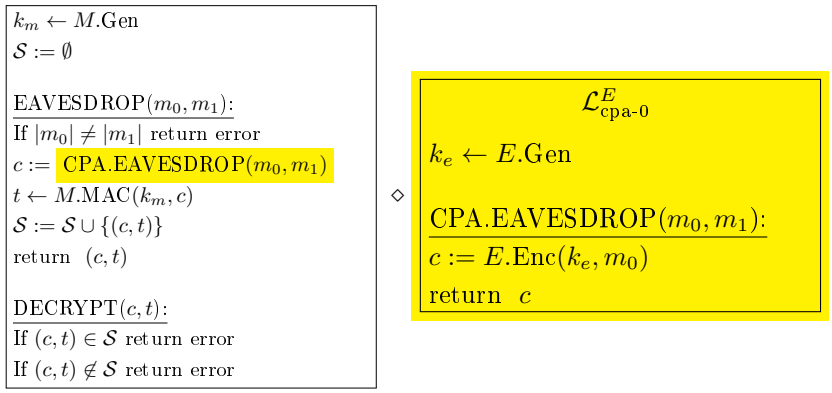
Por lo tanto, las dos condiciones "$(c,t) \in \mathcal{S}$" y "$(c,t) \notin \mathcal{T}$" en la subrutina DEC son ¡exhaustivas! La línea final de DEC es inalcanzable. Esta librería destaca la idea intuitiva de que un adversario puede consultar DEC con un texto cifrado generado por EAVESDROP (el caso $(c,t) \in \mathcal{S}$) - en cuyo caso la respuesta es nula - o con un texto cifrado diferente - en cuyo caso la respuesta será error ya que el MAC no se verificará.

$k_e \leftarrow E.\text{Gen}$
$k_m \leftarrow M.\text{Gen}$
$\mathcal{S} := \emptyset$

EAVESDROP(m_0, m_1):
If $|m_0| \neq |m_1|$ return error
$c := E.\text{Enc}(k_e, m_0)$
$t \leftarrow M.\text{MAC}(k_m, c)$
$\mathcal{S} := \mathcal{S} \cup \{(c,t)\}$
return (c,t)

DECRYPT(c,t):
If $(c,t) \in \mathcal{S}$ return error
If $(c,t) \notin \mathcal{S}$ return error
\\ inalcanzable

Se ha eliminado la sentencia inalcanzable y se han unificado las variables redundantes $\mathcal{S}$ y $\mathcal{T}$. Nótese que esta librería nunca usa E.Dec, lo que hace posible expresar su uso del esquema de cifrado E en términos de $\mathcal{L}_{\text{cpa-0}}$.



Las sentencias que involucran al esquema de cifrado E se han factorizado en términos de $\mathcal{L}_{\text{cpa-0}}$.

Ahora hemos llegado al punto medio de la prueba. La prueba continúa reemplazando $\mathcal{L}^E_{\text{cpa-0}}$ con $\mathcal{L}^E_{\text{cpa-1}}$ (de modo que se cifra m_1 en lugar de m_0), aplicando las mismas modificaciones que antes (pero en orden inverso), para llegar finalmente a $\mathcal{L}^{\text{EtM}}_{\text{cca-1}}$. Los detalles repetitivos se han omitido, pero mencionamos que al enumerar los mismos pasos en orden inverso, los cambios parecen bastante extraños. Por ejemplo, agregamos una sentencia inalcanzable a la subrutina DEC; creamos una variable redundante $\mathcal{T}$ cuyo contenido es el mismo que $\mathcal{S}$; misteriosamente cambiamos una instancia de $\mathcal{S}$ (la condición de la segunda sentencia if en DEC) para referirnos a la otra variable $\mathcal{T}$. Por supuesto, todo esto es para que podamos factorizar las sentencias que se refieren al esquema MAC (junto con $\mathcal{T}$) en términos de $\mathcal{L}^M_{\text{mac-fake}}$ y finalmente reemplazar $\mathcal{L}^M_{\text{mac-fake}}$ con $\mathcal{L}^M_{\text{mac-real}}$.

■

Capítulo 13

Funciones Hash

Supón que compartes un archivo enorme con un amigo, pero no estás seguro de si ambos tienen la misma versión del archivo. Podrías enviar su versión del archivo a tu amigo y él podría compararla con tu versión. ¿Existe alguna forma de verificar que implique menos comunicación que esto?

Llamemos a tu versión del archivo x (una cadena) y a la versión de tu amigo y. El objetivo es determinar si $x = y$. Un enfoque natural es acordar alguna función determinista H, calcular $H(x)$ y enviársela a tu amigo. Tu amigo puede calcular $H(y)$ y, dado que H es determinista, comparar el resultado con tu $H(x)$. Para que este método sea infalible, necesitamos que H tenga la propiedad de que entradas diferentes siempre se mapeen a salidas diferentes; en otras palabras, H debe ser *inyectiva* (uno a uno). Desafortunadamente, si $H : \{0,1\}^{\text{in}} \to \{0,1\}^{\text{out}}$ es inyectiva, entonces out $\geq$ in. ¡Esto significa que enviar $H(x)$ no es mejor/más corto que enviar x en sí mismo!

Llamemos al par (x, y) una *colisión* en H si $x \neq y$ y $H(x) = H(y)$. Una función inyectiva no tiene colisiones. Un tema común en criptografía es que no siempre necesitas que algo sea imposible; a menudo es suficiente que sea muy poco probable. En lugar de decir que H no debería tener colisiones, ¿qué pasaría si simplemente dijéramos que las colisiones deberían ser difíciles (para algoritmos de tiempo polinómico) de encontrar? Una H con esta propiedad probablemente sea lo suficientemente buena para cualquier cosa que nos preocupe. ¡También podría ser posible construir tal H con salidas que sean más cortas que sus entradas!

Lo que hemos estado describiendo es exactamente una *función hash criptográfica*. Una función hash tiene entradas largas y salidas cortas, típicamente $H : \{0,1\}^* \to \{0,1\}^n$. Tal H necesariamente debe tener muchas colisiones. La propiedad de seguridad de una función hash es que es difícil encontrar tal colisión. Otro buen nombre para una función hash (que acabo de inventar y que nadie más usa) sería una función "pseudo-inyectiva". Aunque no es inyectiva, se comporta como una para nuestros propósitos.

13.1. Propiedades de seguridad para funciones hash

Hay tres propiedades de seguridad comunes para funciones hash:

- **Resistencia a colisiones.** Debería ser computacionalmente difícil encontrar cualquier colisión $x \neq x'$ tal que $H(x) = H(x')$.
- **Resistencia a segunda preimagen.** Dado x, debería ser computacionalmente difícil encontrar cualquier colisión que involucre a x. En otras palabras, debería ser difícil computar $x' \neq x$ tal que $H(x) = H(x')$.
- **Resistencia a preimagen.** Dada una imagen y en el rango de H, debería ser computacionalmente difícil encontrar cualquier x tal que $y = H(x)$.

13.1.1. Ataques de fuerza bruta a funciones hash

Existe una diferencia importante entre la resistencia a colisiones, la resistencia a segunda preimagen y la resistencia a preimagen, que se refleja en la dificultad de sus respectivos ataques de fuerza bruta. Supongamos que H es una función hash cuyas salidas tienen una longitud de n bits. Hagamos una suposición simplificadora de que para cualquier $m > n$ la siguiente distribución es aproximadamente uniforme sobre $\{0,1\}^n$:

$x \leftarrow \{0,1\}^m$
return $H(x)$

Esta es una suposición bastante realista para las funciones hash prácticas. Si esto no fuera cierto, entonces H introduciría algún sesgo hacia algunas salidas y lejos de otras salidas, lo que se percibiría como sospechoso. Además, a medida que la salida de H se desvía más de una distribución uniforme, solo facilita la búsqueda de colisiones.

A continuación, se muestran ataques sencillos de fuerza bruta para la resistencia a preimagen (a la izquierda), la resistencia a segunda preimagen (al centro) y resistencia a colisiones (a la derecha):

Fuerza bruta a preimagen

$\underline{\mathcal{A}_{\text{pi}}(y)}$:
while true
  $x \leftarrow \{0,1\}^m$
  if $y = H(x)$ return x

Fuerza bruta a segunda preimagen

$\underline{\mathcal{A}_{\text{2pi}}(x)}$:
$x' := \mathcal{A}_{\text{pi}}(H(x))$
if $x' \neq x$ return x'

Fuerza bruta a colisiones

$\underline{\mathcal{A}_{\text{cr}}()}$:
$x \leftarrow \{0,1\}^m$
$x' := \mathcal{A}_{\text{2pi}}(x)$
return (x, x')

Existe una diferencia fundamental en lo difícil que es romper la resistencia a colisiones, la resistencia a segunda preimagen y la resistencia a preimagen. Romper la resistencia a colisiones es como invitar a más personas a una habitación hasta que la habitación contenga a 2 personas con el mismo cumpleaños. Romper la resistencia a segunda preimagen es como invitar a más personas a la habitación hasta que la habitación contenga a otra persona con *tu* cumpleaños. Romper la resistencia a preimagen es como invitar a más personas a la habitación hasta que la habitación contenga a una persona cuyo cumpleaños sea una fecha predeterminada. En el caso de la resistencia a colisiones, no tienes información sobre la gente que entra a la habitación. En el caso de resistencia a segunda preimagen, puedes usar información adicional sobre ti para encontrar a otra persona con tu mismo cumpleaños. En el caso de resistencia a preimagen, no tienes esa información adicional.

Es interesante señalar que estas tres propiedades no son independientes, sino que hay algunas *más débiles* que otras. Cuando decimos que la propiedad P es *más débil* que la propiedad Q, queremos significar que siempre que se cumple Q también se cumple P, pero P podría cumplirse sin hacerlo Q. Se dice también, si esto es así, que Q es *más fuerte* que P o que Q *implica* a P. En este caso la resistencia a preimagen es más débil que la resistencia a segunda preimagen, que, a su vez, es más débil que la resistencia a colisiones.

resistencia a colisiones $\implies$ resistencia a segunda preimagen $\implies$ resistencia a preimagen

Esto se debe a que los retos a los que se enfrenta el adversario van en orden decreciente de dificultad. Es bastante fácil convercerse de que, si un adversario es capaz de encontrar preimágenes, también podrá encontrar segunda preimágenes. Y de que, si puede encontrar segunda preimágenes, también puede encontar colisiones. Esto queda reflejado en los atáques de fuerza bruta descritos anteriormente donde $\mathcal{A}_{\text{cr}}$ utiliza $\mathcal{A}_{\text{2pi}}$ que, a su vez, utiliza $\mathcal{A}_{\text{pi}}$. Por tanto, aparece una pauta muy común en las definiciones de seguridad en Criptografía:

el problema "más fácil" de resolver para el adversario (que en este caso es encontrar colisiones) da lugar a la noción de seguridad más fuerte o robusta (ya que, si un adversario no es capaz de resolver un problema más fácil, tampoco podrá resolver los que son más difíciles.)

Por todo lo que hemos comentado, es fácil entender que cuando se utiliza una función hash como primitiva en un esquema o protocolo en el que se le requieren las tres propiedades, es muy frecuente simplemente exigir que tenga la propiedad de resistencia a colisiones, ya que, al ser ésta la más fuerte, se consigue automáticamente que las otras dos también se cumplan.

13.1.2. Seguridad de la función hash en la práctica

Nos centraremos en desarrollar una definición formal para la resistencia a colisiones. Podemos inspirarnos en la definición de seguridad para los MAC. La seguridad para un MAC significa que debería ser difícil producir una falsificación. La definición de seguridad del MAC formalizó esa idea con una librería que verifica una falsificación y otra librería que asume que una falsificación es imposible. Si las dos librerías son indistinguibles, entonces debe ser difícil encontrar una falsificación.

Podemos adoptar un enfoque similar para decir que debería ser difícil producir una colisión. Aquí hay un intento:

$$\boxed{\begin{array}{l}\underline{\text{TEST}(x, x'):} \\ \text{if } x \neq x' \text{ and } H(x) = H(x') \text{ return true} \\ \text{else return false}\end{array}} \quad \approx \quad \boxed{\begin{array}{l}\underline{\text{TEST}(x, x'):} \\ \text{return false}\end{array}}$$

Esto corresponde a lo que yo llamaría la "definición popular" de resistencia a colisiones. Tiene sentido intuitivo (siempre y cuando te sientas cómodo con nuestro estilo de definición de seguridad), pero desafortunadamente la definición sufre de un problema técnico muy sutil.

Debido al principio de Kerckhoffs, permitimos que los ataques dependan arbitrariamente del código fuente de las dos librerías. Esta es una forma de formalizar la idea de que "el atacante sabe todo sobre los

algoritmos". Nuestras definiciones de seguridad restringen los ataques a ser algoritmos de tiempo polinómico, ¡pero nunca consideran el esfuerzo que se dedica a encontrar el código fuente del ataque!

Esta extraña laguna conduce al siguiente ataque válido. Cuando consideramos la seguridad de alguna función H, sabemos que existen muchas colisiones (x, x') en H. Estas colisiones pueden ser difíciles de encontrar, pero ciertamente existen. Con tiempo exponencial, podríamos encontrar tal par (x, x') y escribir el código de un atacante:

$\mathcal{A}$

return $\text{TEST}(x, x')$

Aquí, los valores x y x' están codificados directamente en $\mathcal{A}$. El algoritmo $\mathcal{A}$ es claramente de tiempo polinómico (de hecho, de tiempo constante). La "laguna" es que la definición considera solo el coste de *ejecutar* el algoritmo $\mathcal{A}$, y no el coste de encontrar el código fuente de $\mathcal{A}$.

La forma en que se evita este tipo de situación en otras definiciones de seguridad es que las librerías tienen cierta aleatoriedad secreta. Si bien se permite que el atacante dependa arbitrariamente del *código fuente* de las librerías, no se le permite depender de la *elección de los resultados para eventos aleatorios* en las librerías, como el muestreo de una clave secreta. Dado que el atacante no puede "prepararse" para la elección aleatoria a la que se enfrentará, no tenemos ataques tan triviales. Por otro lado, estas dos librerías para la resistencia a colisiones son totalmente deterministas. No hay "sorpresas" sobre para qué función H se le pedirá al atacante que calcule una colisión, por lo que no hay nada que impida que un atacante esté "preparado" con una colisión precalculada en H.

13.1.3. Seguridad de la función hash en teoría

La forma de evitar este problema técnico es introducir cierta aleatoriedad en las librerías y en las entradas de H. Definimos las funciones hash para que tomen dos argumentos: un valor público elegido aleatoriamente s llamado **sal**, y una entrada x elegida por el adversario.

Definición 13.1.1. *Una función hash H es **resistente a colisiones** si $\mathcal{L}^H_{\text{cr-real}} \approx \mathcal{L}^H_{\text{cr-fake}}$, donde:*

$\mathcal{L}^H_{\text{cr-real}}$

$s \leftarrow \{0,1\}^\lambda$

GETSALT():
return s

TEST($x, x' \in \{0,1\}^*$):
if $x \neq x'$ and $H(s,x) = H(s,x')$ return true
return false

$\approx$

$\mathcal{L}^H_{\text{cr-fake}}$

$s \leftarrow \{0,1\}^\lambda$

GETSALT():
return s

TEST($x, x' \in \{0,1\}^*$):
return false

La librería inicialmente muestrea la sal s. A diferencia de otras librerías, este valor s está destinado a ser proporcionado al atacante, por lo que la librería proporciona una forma (GETSALT) para que el atacante lo obtenga. El atacante luego intenta encontrar una colisión $x \neq x'$ donde $H(s,x) = H(s,x')$.

No sé por qué se usa el término "sal" con las funciones hash. La razón parece ser un misterio para Internet.[1] Piensa en la sal como un valor extra que "personaliza" la función hash para una aplicación dada. Aquí hay una buena analogía: un esquema de cifrado puede considerarse como un algoritmo de cifrado diferente $\text{Enc}(k, \cdot)$ para cada elección de clave k. Cuando elijo una k aleatoria, obtengo un algoritmo de cifrado personalizado $\text{Enc}(k, \cdot)$ que no está relacionado con el algoritmo $\text{Enc}(k', \cdot)$ que otra persona obtendría cuando elige su propia k'. Cuando elijo una sal s, obtengo una función hash personalizada $H(s, \cdot)$ que no está relacionada con otras funciones $H(s', \cdot)$. Debido a que la sal se elige uniformemente de $\{0,1\}^\lambda$, un programa que llama no puede predecir con qué sal (qué función hash personalizada) será desafiado.

La Definición 13.1.1 es una definición válida para la resistencia a colisiones, libre de extrañas lagunas como la definición "popular". Sin embargo, no es una definición particularmente útil para usar en pruebas de seguridad, cuando una función hash se usa como un bloque de construcción en un sistema más grande. Se vuelve engorroso usarla en esos casos, porque cuando usas una función hash, normalmente no verificas *explícitamente* si has visto una colisión. En cambio, simplemente procedes como si las colisiones no fueran a suceder.

En este capítulo, no veremos declaraciones de seguridad demostrables que se refieran a esta definición.

13.1.4. Sal en la práctica

Cuando definimos las funciones hash en teoría, requerimos que la función hash acepte dos entradas, la primera de las cuales se interpreta como sal. Las funciones hash que ves en la práctica solo tienen una entrada, una cadena de longitud arbitraria. Puedes simular el efecto de una sal para tal función hash simplemente concatenando las dos entradas, p. ej., $H(s||x)$ en lugar de $H(s, x)$.

El concepto de un hash con sal no solo es útil para hacer una definición de seguridad coherente, sino que también es una buena práctica. Las funciones hash se usan comúnmente para almacenar contraseñas. Un servidor puede almacenar registros de usuario de la forma (nombre de usuario, $h = H(\text{contraseña})$). Cuando un usuario intenta iniciar sesión con la contraseña p', el servidor calcula $H(p')$ y lo compara con h. Almacenar contraseñas con hash significa que, en caso de que se robe el archivo de contraseñas, un atacante necesitaría encontrar una preimagen de h para suplantar al usuario.

La mejor práctica es usar una sal separada para cada usuario. En lugar de almacenar (nombre de usuario, $H(\text{contraseña})$), elige una sal aleatoria s para cada usuario y almacena (nombre de usuario, $s, H(s||\text{contraseña})$). Las propiedades de seguridad de una función hash no requieren que s sea secreta, aunque tampoco hay una buena razón para difundir públicamente la sal de un usuario. La sal solo la necesita el servidor, cuando verifica una contraseña durante un intento de inicio de sesión.

Una sal específica del usuario significa que cada usuario obtiene su propia función hash "personalizada" para almacenar su contraseña. La sal ofrece los siguientes beneficios:

- Sin sal, sería evidente cuando dos usuarios tienen la misma contraseña: tendrían los mismos hashes de contraseña. La misma contraseña con hash con diferentes sales dará como resultado salidas hash no relacionadas.

[1] Si tienes un argumento aleatorio adicional para una función hash, pero lo mantienes secreto, se llama "pimienta". Lo digo en serio, esto es real.

- Un atacante puede calcular un diccionario de $(p, H(p))$ para contraseñas comunes. Sin sal, este diccionario facilita el ataque a todos los usuarios a la vez, ya que todos los usuarios están usando la misma función hash. Con sal, cada usuario tiene una función hash personalizada, cada una de las cuales requeriría su propio diccionario. La sal hace que el esfuerzo de un atacante escale con el número de víctimas.

Capítulo 14

RSA y firmas digitales

RSA fue uno de los primeros criptosistemas de clave pública desarrollados. Se describió por primera vez en 1978 y lleva el nombre de sus creadores, Ron Rivest, Adi Shamir y Len Adleman.[1] RSA se puede utilizar como un bloque de construcción para el cifrado de clave pública y las firmas digitales. En este capítulo, discutimos solo la aplicación de RSA para firmas digitales.

14.1. División módulo n

Por favor, revisa el material de la Sección 1.2, para asegurarte de que tu comprensión de la aritmética modular básica esté fresca. Debes sentirte cómodo con las definiciones de $\mathbb{Z}_n$, congruencia ($\equiv_n$), el operador módulo (mód) y cómo realizar sumas, multiplicaciones y restas módulo n.

Ten en cuenta que no hemos mencionado la *división* módulo n. ¿Tiene siquiera sentido hablar de división módulo n?

Ejemplo 14.1. Considere los siguientes hechos que se cumplen módulo 15:

$$\begin{aligned} 2 \times 8 &\equiv_{15} 1 \\ 4 \times 8 &\equiv_{15} 2 \\ 6 \times 8 &\equiv_{15} 3 \\ 8 \times 8 &\equiv_{15} 4 \\ 10 \times 8 &\equiv_{15} 5 \\ 12 \times 8 &\equiv_{15} 6 \\ 14 \times 8 &\equiv_{15} 7 \end{aligned}$$

[1] Clifford Cocks desarrolló un esquema equivalente en 1973, pero fue clasificado porque trabajaba para la inteligencia británica.

Ahora imagine reemplazar "$\times 8$" con "$\div 2$" en cada uno de estos ejemplos:

$$\begin{aligned} 2 \div 2 &\equiv_{15} 1 \\ 4 \div 2 &\equiv_{15} 2 \\ 6 \div 2 &\equiv_{15} 3 \\ 8 \div 2 &\equiv_{15} 4 \\ 10 \div 2 &\equiv_{15} 5 \\ 12 \div 2 &\equiv_{15} 6 \\ 14 \div 2 &\equiv_{15} 7 \end{aligned}$$

¡Todo sigue teniendo sentido! De alguna manera, multiplicar por 8 mod 15 parece ser lo mismo que "dividir por 2" mod 15.

Los ejemplos anteriores usaron $x \cdot 8$ $(x \div 2)$ donde x era un número par. ¿Qué sucede cuando x es un número impar?

$$3 \cdot 8 \equiv_{15} 9 \iff \text{“}3 \div 2 \equiv_{15} 9\text{”} \text{ ??}$$

Esto podría parecer sin sentido, pero si hacemos las sustituciones $3 \equiv_{15} -12$ y $9 \equiv_{15} -6$, entonces obtenemos algo que sí tiene sentido:

$$-12 \cdot 8 \equiv_{15} -6 \iff -12 \div 2 \equiv_{15} -6$$

Este ejemplo muestra que seguramente existe alguna relación interesante entre los números 2, 8 y 15. Parece razonable interpretar "multiplicación por 8" como "división por 2" cuando se trabaja mod 15.

¿Hay alguna manera de que podamos hacer algo similar para la "división por 3" mod 15? ¿Podemos encontrar algún y donde la "multiplicación por y mod 15" tenga el mismo comportamiento que la "división por 3 mod 15"? En particular, buscaríamos un valor y que satisfaga $3 \cdot y \equiv_{15} 1$, pero puedes comprobar por ti mismo que **no such value of y exists**.

¿Por qué podemos "dividir por 2" mod 15 pero aparentemente no podemos "dividir por 3" mod 15? Exploraremos esta pregunta en el resto de esta sección.

14.1.1. Inversos Multiplicativos

Normalmente no usamos directamente la terminología de "división" en la aritmética modular. En lugar de decir "división entre 2", decimos "multiplicación por 2^{-1}", donde 2^{-1} es simplemente otro nombre para 8.

Definición 14.1.1 ($x^{-1} \mod n$)**.** *El **inverso multiplicativo** de $x \mod n$ es el entero y que satisface $x \cdot y \equiv_n 1$ (si tal número existe). Normalmente nos referimos al inverso multiplicativo de x como "x^{-1}".*

Ejemplo 14.2. Continuando trabajando módulo 15, tenemos:

- $4^{-1} \equiv_{15} 4$ ya que $4 \cdot 4 = 16 \equiv_{15} 1$. Por lo tanto, ¡4 es su propio inverso multiplicativo! También puedes entender esto como: $4^{-1} = (2^2)^{-1} = (2^{-1})^2 \equiv_{15} 8^2 = 64 \equiv_{15} 4$.

- $7^{-1} \equiv_{15} 13$ ya que $7 \cdot 13 = 91 \equiv_{15} 1$.

Estamos interesados en qué números tienen un inverso multiplicativo mod n.

Definición 14.1.2 ($\mathbb{Z}_n^*$)**.** *El* ***grupo multiplicativo***[2] *módulo n se define como:*

$$\mathbb{Z}_n^* = \{x \in \mathbb{Z}_n \mid x \text{ tiene un inverso multiplicativo mod } n\}$$

Por ejemplo, hemos visto que $\mathbb{Z}_{15}^*$ contiene los números 2, 4 y 7 (y quizás otros), pero no contiene el número 3 ya que 3 no tiene un inverso multiplicativo.

Entonces, ¿qué números tienen un inverso multiplicativo mod n, en general? (¿Qué números pertenecen a $\mathbb{Z}_n^*$?) La respuesta es bastante simple:

Teorema 14.1.3. *x tiene un inverso multiplicativo mod n* ***si y solo si*** *$MCD(x, n) = 1$. En otras palabras, $\mathbb{Z}_n^* = \{x \in \mathbb{Z}_n \mid MCD(x, n) = 1\}$.*

Demostramos el teorema usando otro hecho del álgebra abstracta que a menudo es útil:

Teorema 14.1.4 (Teorema de Bézout)**.** *Para todos los enteros x e y, existen enteros a y b tales que $ax + by = MCD(x, y)$. De hecho, $MCD(x, y)$ es el entero positivo más pequeño que se puede escribir como una combinación lineal entera de x e y.*

No demostraremos el teorema de Bézout, pero mostraremos cómo se usa para demostrar el Teorema 14.1.3:

Demostración. (del Teorema 14.1.3) ($\Leftarrow$) Supongamos que $\text{MCD}(x, n) = 1$. Mostraremos que x tiene un inverso multiplicativo mod n. Del Teorema de Bézout, existen enteros a, b que satisfacen $ax + bn = 1$. Al reducir ambos lados de esta ecuación módulo n, tenemos

$$1 = ax + bn \equiv_n ax + b \cdot 0 = ax.$$

Por lo tanto, el entero a que sale del teorema de Bézout es el inverso multiplicativo de x módulo n.

($\Rightarrow$) Supongamos que x tiene un inverso multiplicativo mod n, entonces $xx^{-1} \equiv_n 1$. Necesitamos mostrar que $\text{MCD}(x, n) = 1$. De la definición de $\equiv_n$, sabemos que n divide a $xx^{-1} - 1$, por lo que podemos escribir $xx^{-1} - 1 = kn$ (como una expresión sobre los enteros) para algún entero k. Reordenando, tenemos $xx^{-1} - kn = 1$. Dado que podemos escribir 1 como una combinación lineal entera de x y n, el teorema de Bézout dice que debemos tener $\text{MCD}(x, n) = 1$.

■

Ejemplo 14.3. $\mathbb{Z}_{15} = \{0, 1, \ldots, 14\}$, y para obtener $\mathbb{Z}_{15}^*$ excluimos cualquiera de los números que comparten un factor común con 15. En otras palabras, excluimos los múltiplos de 3 y los múltiplos de 5. Los números restantes son $\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$.

Dado que 11 es primo, 0 es el único número en $\mathbb{Z}_{11}$ que comparte un factor común con 11. Todos los demás satisfacen $\text{MCD}(x, 11) = 1$. Por lo tanto, $\mathbb{Z}_{11}^* = \{1, 2, \ldots, 10\}$.

[2] "Grupo" es un término técnico del álgebra abstracta.

La relación entre los inversos multiplicativos y el MCD va incluso más allá del Teorema 14.1.3. Recuerda que podemos calcular $\text{MCD}(x, n)$ eficientemente utilizando el algoritmo de Euclides. Existe una modificación relativamente simple al algoritmo de Euclides que también calcula los coeficientes de Bézout correspondientes con poco trabajo extra. En otras palabras, dados x y n, es posible calcular eficientemente enteros a, b y d tales que

$$ax + bn = d = \text{MCD}(x, n)$$

En el caso donde $\text{MCD}(x, n) = d = 1$, el entero a es un inverso multiplicativo de x mod n. El "algoritmo extendido de Euclides" para MCD se da a continuación:

```
EXTMCD(x, y):
// devuelve (d, a, b) tal que MCD(x, y) = d = ax + by
if y = 0:
    return (x, 1, 0)
else
    (d, a, b) := EXTMCD(y, x  mód y)
    return (d, b, a − b⌊x/y⌋)
```

14.1.2. La función ϕ de Euler

La función ϕ de Euler se define como $\phi(n) \stackrel{\text{def}}{=} |\mathbb{Z}_n^*|$; es decir, el número de elementos de $\mathbb{Z}_n$ que tienen inversos multiplicativos.

Como ejemplo, si n es un primo, entonces $\mathbb{Z}_n^* = \mathbb{Z}_n \setminus \{0\}$ porque cada entero en $\mathbb{Z}_n$ aparte del cero es relativamente primo con n. Por lo tanto, $\phi(n) = n - 1$ en este caso.

El RSA involucra un módulo n que es el producto de dos primos distintos $n = pq$. En ese caso, $\phi(n) = (p-1)(q-1)$. Para ver por qué, contemos cuántos elementos en $\mathbb{Z}_{pq}$ comparten un divisor común con pq (es decir, no están en $\mathbb{Z}_{pq}^*$).

- Los múltiplos de p comparten un divisor común con pq. Estos incluyen $0, p, 2p, 3p, ..., (q-1)p$. Hay q elementos en esta lista.
- Los múltiplos de q comparten un divisor común con pq. Estos incluyen $0, q, 2q, 3q, ..., (p-1)q$. Hay p elementos en esta lista.

Hemos contado claramente doblemente el elemento 0 en estas listas. Pero ningún otro elemento se cuenta doblemente. Cualquier elemento que aparezca en ambas listas sería un múltiplo común tanto de p como de q, pero dado que p y q son relativamente primos, su mínimo común múltiplo es pq, que es mayor que cualquier elemento en estas listas.

Contamos $p + q - 1$ elementos de $\mathbb{Z}_{pq}$ que comparten un divisor común con pq. El resto pertenece a $\mathbb{Z}_{pq}^*$, y hay $pq - (p + q - 1) = (p-1)(q-1)$ de ellos. Hence $\phi(pq) = (p-1)(q-1)$.

Existen fórmulas generales para $\phi(n)$, pero normalmente dependen de conocer la factorización prima de n. Veremos más conexiones entre la dificultad de calcular $\phi(n)$ y la dificultad de factorizar n más adelante en esta parte del curso.

La razón por la que consideramos $\phi(n)$ es este teorema fundamental del álgebra abstracta:

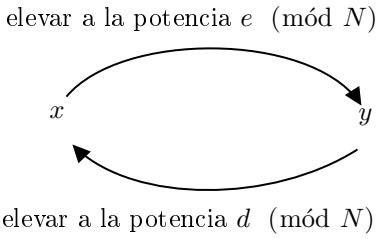
Teorema 14.1.5 (Teorema de Euler)**.** *Si* $x \in \mathbb{Z}_n^*$*, entonces* $x^{\phi(n)} \equiv_n 1$.

14.2. La función RSA

La función RSA se define de la siguiente manera:

- Sean p y q primos distintos (más adelante diremos más sobre cómo se eligen), y sea $N = pq$. N se llama el *módulo RSA*.
- Sean e y d enteros tales que $ed \equiv_{\phi(N)} 1$. Es decir, e y d son inversos multiplicativos mód $\phi(N)$ - ¡no mód N!
- La función RSA es: $x \mapsto x^e \mod N$, donde $x \in \mathbb{Z}_N$.
- La función RSA inversa es: $y \mapsto y^d \mod N$, donde $y \in \mathbb{Z}_N$.

Esencialmente, la función RSA (y su inversa) es una simple exponenciación modular. Lo más confuso para recordar sobre RSA es que e y d "viven" en $\mathbb{Z}_{\phi(N)}^*$, mientras que x e y "viven" en $\mathbb{Z}_N$.



Podemos argumentar que elevar a la potencia e y elevar a la potencia d son inversas en general: Dado que $ed \equiv_{\phi(N)} 1$, podemos escribir $ed = t\phi(N) + 1$ para algún entero t. Entonces:

$$(x^e)^d = x^{ed} = x^{t\phi(N)+1} = (x^{\phi(N)})^t x \equiv_N 1^t x = x$$

Ten en cuenta que hemos utilizado el hecho de que $x^{\phi(N)} \equiv_N 1$ del Teorema de Euler.

14.2.1. Cómo [No] exponenciar números enormes

Cuando ves una expresión como "$x^e \mod N$", podrías verte tentado a implementarla con el siguiente algoritmo:

```
NAIVEEXPONENTIATE(x, e, N):
result = 1
for i = 1 to e: \\ calcular x^e
    result := result×x
return result  mód N
```

Si bien este algoritmo daría la respuesta correcta, es una forma realmente mala de hacerlo. En la práctica, usamos RSA con números que tienen miles de bits de longitud. Supongamos que ejecutamos el algoritmo NAIVEEXPONENTIATE con argumentos x, e y N que tienen alrededor de mil bits cada uno (por lo que la magnitud de estos números es cercana a 2^{1000}):

1. ¡El algoritmo pasará aproximadamente 2^{1000} iteraciones en el bucle `for`!
2. El algoritmo calcula x^e como un entero primero, y luego reduce ese entero mod N. Observe que x^2 tiene aproximadamente 2000 bits de longitud, x^3 tiene aproximadamente 3000 bits de longitud, etc. Por lo que se necesitarían alrededor de $2^{1000} \cdot 1000$ bits solo para escribir el entero x^e.

Como puedes ver, no hay suficiente tiempo ni capacidad de almacenamiento en el universo para usar este algoritmo. Entonces, ¿cómo podemos calcular realmente valores como $x^e \mod N$ con números enormes?

1. Supón que te dan un entero x y te piden calcular x^{17}. Puedes calcularlo como:

$$x^{17} = \underbrace{x \cdot x \cdot x \cdot \cdots \cdot x}_{16 \text{ multiplicaciones}}$$

   Pero una forma más inteligente es observar que:

$$x^{17} = x^{16} \cdot x = (((x^2)^2)^2)^2 \cdot x.$$

   Esta expresión se puede evaluar con solo 5 multiplicaciones (elevar al cuadrado es simplemente multiplicar un número por sí mismo).

   De forma más general, puedes calcular una expresión como x^e siguiendo la recurrencia que se muestra a continuación. El método se llama **exponenciación binaria** (o por cuadrados repetidos), por razones que, con suerte, son claras:

$$x^e = \begin{cases} 1 & \text{si } e = 0 \\ (x^{\frac{e}{2}})^2 & \text{si } e \text{ es par} \\ (x^{\frac{e-1}{2}})^2 \cdot x & \text{si } e \text{ es impar} \end{cases}$$

```
BETTEREXP(x, e):
if e = 0: return 1
if e es par:
    return BETTEREXP(x, e/2)^2
if e es impar:
    return BETTEREXP(x, (e-1)/2)^2 · x
```

   BETTEREXP divide el argumento e entre dos (más o menos) cada vez que se llama recursivamente, hasta llegar al caso base. Por lo tanto, el número de llamadas recursivas es $\mathcal{O}(\log e)$. En cada llamada recursiva solo hay un número constante de multiplicaciones (incluidas las elevaciones al cuadrado). Así que, en general, este algoritmo requiere solo $\mathcal{O}(\log e)$ multiplicaciones (comparado con $e - 1$ multiplicaciones al simplemente multiplicar m por sí mismo e veces). En el caso donde $e \sim 2^{1000}$, esto significa solo unos pocos miles de multiplicaciones.

2. Nos importa solo $x^e \mod N$, no el valor entero intermedio x^e. Una de las características más fundamentales de la aritmética modular es que puedes reducir cualquier valor intermedio mod N si solo te importa el resultado final mod N.

Revisitando nuestro ejemplo anterior:

$$x^{17} \mod N = x^{16} \cdot x \mod N = (((x^2 \mod N)^2 \mod N)^2 \mod N)^2 \cdot x \mod N.$$

De forma más general, podemos reducir todos los valores intermedios mod N:

MODEXP(x, e, N): \\ calcula $x^2 \mod N$
if $e = 0$: return 1
if e es par:
    return BETTEREXP$(x, \frac{e}{2})^2 \mod N$
if e es impar:
    return BETTEREXP$(x, \frac{e-1}{2})^2 \cdot x \mod N$

Este algoritmo evita el problema de calcular el entero astronómicamente grande x^e. Nunca necesita almacenar ningún valor (mucho) mayor que N.

Advertencia: Incluso este algoritmo MODEXP no es una forma ideal de implementar la exponenciación con fines criptográficos.

14.2.2. Propiedades de seguridad y discusión

RSA es lo que se llama una *función trampa*.

- Un usuario genera los parámetros RSA (principalmente N, e, y d) y hace públicos N y e, mientras mantiene d en privado.
- Propiedades de funcionalidad: Dada solo la información pública N y e, es fácil calcular la función RSA ($x \mapsto x^e \mod N$). Dada la información privada (d) es claramente fácil calcular la inversa RSA ($y \mapsto y^d \mod N$).
- **Propiedad de seguridad**: Dada solo la información pública, debería ser difícil calcular la inversa RSA ($y \mapsto y^d \mod N$) sobre valores elegidos aleatoriamente. En otras palabras, la única persona que puede calcular la función inversa RSA es la persona que generó los parámetros RSA.

Actualmente los mejores ataques conocidos contra RSA (i.e., las formas de calcular la función inversa RSA dada solo la información pública) implican factorizar el módulo. Si queremos asegurar que RSA sea seguro como una función trampa, debemos comprender el estado del arte para la factorización de números grandes.

Antes de discutir el rendimiento de los algoritmos de factorización, recuerda que medimos el rendimiento como una función de la longitud de la entrada: cuántos bits se necesitan para escribir la entrada. En un algoritmo de factorización, la entrada es un número grande N, y se necesitan aproximadamente $n = \log_2 N$ bits para escribir ese número. Discutiremos el tiempo de ejecución de los algoritmos como una función de n, no de N. Simplemente ten en cuenta la diferencia en el costo entre escribir un número de 1000 bits ($n = 1000$) frente a contar hasta un número de 1000 bits ($N = 2^{1000}$).

Todo el mundo conoce el método de "división por prueba" para la factorización: dado un número N, verificar si i divide a N, para cada $i \in \{2, ..., \sqrt{N}\}$. Este algoritmo requiere $\sqrt{N} = 2^{n/2}$ divisiones en el

peor de los casos. Es un algoritmo de tiempo exponencial ya que medimos el rendimiento en términos de la longitud en bits n.

Si este fuera el mejor algoritmo de factorización conocido, entonces necesitaríamos hacer que N sea tan grande como 2^{256} para que la factorización requiera un esfuerzo de 2^{128}. Pero hay algoritmos de factorización mucho mejores que la división por prueba. El algoritmo de factorización más rápido en la actualidad se llama Criba General del Cuerpo de Números (GNFS), y su complejidad es algo así como

$$O\left(n^{\left(\frac{n}{\log n}\right)^{\frac{1}{3}}}\right)$$

no es un algoritmo de tiempo polinómico, pero es mucho más rápido que la división por prueba.

A partir de enero de 2020, el módulo RSA más grande que se ha factorizado (públicamente) es un módulo de 795 bits. Factorizar este número requirió el equivalente a 900 años de núcleo de CPU, o aproximadamente 2^{66} ciclos de reloj totales.

Todo esto quiere decir que los números involucrados en RSA deben ser bastante grandes para resistir ataques de factorización (es decir, requieren un esfuerzo de 2^{128} para los algoritmos de factorización de última generación). Las mejores prácticas actuales sugieren usar módulos RSA de 2048 o 4096 bits, lo que significa que p y q tienen cada uno 1024 o 2048 bits.

14.3. Firmas digitales

Los MACs son una primitiva criptográfica que proporciona autenticidad. Una etiqueta MAC válida en m es una "prueba" de que alguien que conoce la clave ha garantizado m. Los MACs son una primitiva de clave simétrica, en el sentido de que generar una etiqueta MAC y verificar una etiqueta MAC requieren la misma clave (de hecho, una etiqueta se verifica volviéndola a calcular).

Las firmas digitales son similares a los MACs, pero con claves separadas para firmar y verificar. Un esquema de firma digital consta de los siguientes algoritmos:

- **Gen**: produce un *par* de claves (sk, vk), donde sk es la clave de firma y vk es la clave de verificación.
- **Sign**: toma la clave de firma sk y un mensaje m como entrada, y produce una *firma* σ.
- **Ver**: toma la clave de verificación vk, el mensaje m y una posible firma σ como entrada; produce un booleano.

Si, de hecho, σ es una salida de $\texttt{Sign}(sk, m)$, entonces $\texttt{Ver}(vk, m, \sigma)$ debería producir verdadero. Intuitivamente, debería ser difícil para un atacante encontrar cualquier otro par (m, σ) que haga que `Ver` produzca verdadero.

La idea detrás de las firmas digitales es hacer pública vk. En otras palabras, cualquiera (incluso el atacante) debería poder verificar las firmas. Pero solo el poseedor de sk (la persona que generó vk y sk) debería poder generar firmas válidas. Además, esta garantía debería mantenerse incluso contra un atacante que ve muchos ejemplos de firmas válidas. El atacante no debería poder generar nuevas firmas válidas.

Formalizamos esta propiedad de seguridad de forma similar a como formalizamos la seguridad de los MACs: "solo el poseedor de la clave secreta puede generar etiquetas válidas, incluso después de ver ejemplos elegidos de etiquetas válidas."

Definición 14.3.1. *Sea Σ un esquema de firma. Decimos que Σ es una* ***firma segura*** *si $\mathcal{L}^{\Sigma}_{sig\text{-}real} \approx \mathcal{L}^{\Sigma}_{sig\text{-}fake}$, donde:*

$\mathcal{L}^{\Sigma}_{sig\text{-}real}$

$(vk, sk) \leftarrow \Sigma.Gen$

GETVK():
return vk

GETSIG(m):
return $\Sigma.\mathtt{Sign}(sk, m)$

VERSIG(m, σ)*:*
return $\Sigma.\mathtt{Ver}(vk, m, \sigma)$

$\mathcal{L}^{\Sigma}_{sig\text{-}fake}$

$(vk, sk) \leftarrow \Sigma.Gen$
$\mathcal{S} := \emptyset$

GETVK():
return vk

GETSIG(m):
$\sigma := \Sigma.\mathtt{Sign}(sk, m)$
$\mathcal{S} := \mathcal{S} \cup \{(m, \sigma)\}$
return σ

VERSIG(m, σ)*:*
return $(m, \sigma) \stackrel{?}{\in} \mathcal{S}$

De forma similar a la definición de seguridad para los MACs, las librerías difieren solo en cómo verifican las firmas proporcionadas por el atacante (VERSIG). Si el atacante puede generar un par mensaje-firma (m, σ) que (1) se verifica correctamente, pero (2) no fue generado previamente por la propia librería, entonces VERSIG de la librería $\mathcal{L}_{\text{sig-real}}$ devolverá verdadero, mientras que la librería $\mathcal{L}_{\text{sig-fake}}$ devolvería falso. Al pedir que las librerías sean indistinguibles, en realidad estamos pidiendo que el atacante no pueda encontrar ningún par mensaje-firma de este tipo (falsificación).

La principal diferencia con la definición de MAC es que, a diferencia del entorno MAC, tenemos la intención de hacer pública una clave de verificación. La librería puede ejecutar $(vk, sk) \leftarrow \mathtt{Gen}$, pero estos valores permanecen privados de forma predeterminada. Para hacer público vk, proporcionamos explícitamente un acceso GETVK al atacante.

14.3.1. Firmas RSA de "libro de texto"

Las firmas tienen una asimetría: todo el mundo debería poder verificar una firma, pero solo el poseedor de la clave de firma debería poder generar una firma válida. La función RSA tiene una asimetría similar: si N y e son públicos, entonces cualquiera puede elevar valores a la potencia e, pero solo alguien con d puede elevar valores a la potencia d.

Esta similitud sugiere que podemos usar RSA para firmas de la siguiente manera:

- La clave de verificación es (N, e) y la clave de firma es (N, d), donde estos valores tienen la relación RSA apropiada.
- Una firma del mensaje m (aquí m es un elemento de $\mathbb{Z}_N$) es el valor $\sigma = m^d \mod N$. Intuitivamente, solo alguien con la clave de firma puede generar este valor para un m dado.
- Para verificar una firma σ en un mensaje m, nuestro objetivo es verificar si $\sigma \equiv_N m^d$. Sin embargo, solo se nos dan N y e, no d. Consideremos elevar ambos lados de esta ecuación a la potencia e:

$$\sigma^e \equiv_N (m^d)^e \equiv_N m$$

La segunda igualdad proviene de la propiedad estándar de RSA. Ahora esta verificación se puede realizar solo con la información pública N y e.

A continuación, se proporciona una descripción formal de este esquema:

Definición 14.3.2 (RSA de Libro de Texto)**.** *El algoritmo de generación de claves no se enumera aquí, pero N, e, d se generan de la forma habitual para RSA. La clave de firma es $sk = (N, d)$ y la clave de verificación es $vk = (N, e)$.*

$\underline{\mathtt{Sign}\Big(sk = (N,d), m\Big):}$	$\underline{\mathtt{Ver}\Big(vk = (N,e), m, \sigma\Big):}$
return $m^d \mod N$	$m' := \sigma^e \mod N$
	return $m' \stackrel{?}{=} m$

Desafortunadamente, las firmas RSA de libro de texto solo son útiles como una primera intuición. **¡No son seguras!** Un ataque simple es el siguiente:

Supongamos que un atacante conoce la clave de verificación (N, e) y ve una firma válida $\sigma \equiv_N m^d$ para algún mensaje m. Entonces σ^2 también es una firma válida para el mensaje m^2, ya que:

$$\sigma^2 \equiv_N (m^d)^2 = (m^2)^d$$

El atacante puede generar fácilmente una firma falsificada en un nuevo mensaje m^2, haciendo que el esquema sea inseguro. Por lo tanto, **nunca se debe utilizar este esquema en la práctica**.

14.3.2. Firmas RSA con hash

El problema con las firmas RSA de libro de texto es que las firmas y los textos planos tenían una relación algebraica muy fuerte. Elevar al cuadrado la firma tenía el efecto de elevar al cuadrado el mensaje subyacente. Una forma de solucionar el problema es "romper" esta relación algebraica. Las firmas RSA con hash rompen la estructura algebraica aplicando la función RSA no a m directamente, sino a $H(m)$, donde H es una función hash adecuada (con salidas interpretadas como elementos de $\mathbb{Z}_N$).

Definición 14.3.3 (RSA con hash)**.**

$\underline{\mathtt{Sign}\Big(sk = (N,d), m\Big):}$	$\underline{\mathtt{Ver}\Big(vk = (N,e), m, \sigma\Big):}$
return $H(m)^d \mod N$	$y := \sigma^e \mod N$
	return $H(m) \stackrel{?}{=} y$

Veamos cómo este cambio frustra el ataque a las firmas de libro de texto. Si σ es una firma válida de m, tenemos $\sigma \equiv_N H(m)^d$. Elevar al cuadrado ambos lados lleva a $\sigma^2 \equiv_N (H(m)^2)^d$. ¿Es esta la firma válida de algún m'? Un atacante tendría que identificar algún m' que tenga $H(m') = H(m)^2$. Si la función hash es buena, entonces esto debería ser difícil.

Por supuesto, esto no es una prueba formal. Es posible probar formalmente la seguridad de las firmas RSA con hash. La afirmación precisa de seguridad es: "si RSA es una función trampa segura y H se modela como un oráculo aleatorio, entonces las firmas RSA con hash son un esquema de firma seguro." Dado que no hemos dado definiciones formales ni para las funciones trampa ni para los oráculos aleatorios, no veremos la prueba en este libro.

14.4. La dificultad de factorizar N

Como se mencionó anteriormente, la mejor forma conocida de romper la seguridad de RSA como una función trampa (i.e., para calcular la función inversa de RSA dada solo la información pública N y e) implica factorizar el módulo RSA.

Se cree que la factorización de enteros (o, más específicamente, la factorización de módulos RSA) es un problema difícil para los ordenadores clásicos. En esta sección mostraremos que algunos otros problemas relacionados con RSA son "tan difíciles como la factorización". ¿Qué significa que un problema computacional sea "tan difícil como la factorización"? Más formalmente, en esta sección mostraremos lo siguiente:

Teorema 14.4.1. *Cualquiera de los siguientes problemas se puede resolver en tiempo polinómico, o ninguno de ellos puede:*

1. *Dado un módulo RSA $N = pq$, calcular sus factores p y q.*
2. *Dado un módulo RSA $N = pq$, calcular $\phi(N) = (p-1)(q-1)$.*
3. *Dado un módulo RSA $N = pq$ y el valor e, calcular el d correspondiente (que satisface $ed \equiv_{\phi(N)} 1$).*
4. *Dado un módulo RSA $N = pq$, encontrar cualquier $x \not\equiv \pm 1$ tal que $x^2 \equiv_N 1$.*

Para probar el teorema, mostraremos:

- si existe un algoritmo eficiente para (1), entonces podemos usarlo como una subrutina para construir un algoritmo eficiente para (2). Esto es sencillo: si tienes una subrutina que factoriza N en p y q, entonces puedes llamar a la subrutina y luego calcular $(p-1)(q-1)$.
- si existe un algoritmo eficiente para (2), entonces podemos usarlo como una subrutina para construir un algoritmo eficiente para (3). Esto también es sencillo: si tienes una subrutina que calcula $\phi(N)$ dado N, entonces puedes calcular d exactamente como se calcula en el algoritmo de generación de claves.
- si existe un algoritmo eficiente para (3), entonces podemos usarlo como una subrutina para construir un algoritmo eficiente para (4).
- si existe un algoritmo eficiente para (4), entonces podemos usarlo como una subrutina para construir un algoritmo eficiente para (1).

A continuación, nos centraremos en las dos últimas implicaciones.

14.4.1. Usando raíces cuadradas de la unidad para factorizar N

El problema (4) del Teorema 14.4.1 se refiere a un nuevo concepto conocido como raíces cuadradas de la unidad:

Definición 14.4.2 (Raíz de la unidad)**.** *Decimos que x es una raíz cuadrada de la unidad módulo N si $x^2 \equiv_N 1$. Si $x \not\equiv_N 1$ y $x \not\equiv_N -1$, entonces decimos que x es una raíz cuadrada de la unidad no trivial.*

Dado que $(\pm 1)^2 = 1$ sobre los enteros, también es cierto que $(\pm 1)^2 \equiv_N 1$. En otras palabras, ± 1 son siempre raíces cuadradas de la unidad módulo N, para cualquier N. Pero algunos valores de N tienen incluso más raíces cuadradas de la unidad. Si N es el producto de primos impares distintos, entonces N tiene 4 raíces cuadradas de la unidad: dos triviales y dos no triviales.

Afirmación 14.4.3. *Supongamos que existe un algoritmo eficiente para calcular raíces cuadradas no triviales de la unidad módulo N. Entonces existe un algoritmo eficiente para factorizar N. (Este es el paso (4) $\Rightarrow$ (1) en el Teorema* (14.4.1)*).*

Demostración. La reducción es bastante simple. Supongamos que `NTSRU` es un algoritmo que, al recibir N como entrada, devuelve una raíz cuadrada no trivial de la unidad módulo N. Entonces podemos factorizar N con el siguiente algoritmo:

```
FACTOR(N):
x := NTSRU(N)
returnMCD(N, x + 1) y MCD(N, x − 1)
```

El algoritmo es simple, pero debemos argumentar que es correcto. Cuando x es una raíz cuadrada no trivial de la unidad módulo N, tenemos lo siguiente:

$$\begin{aligned} x^2 \equiv_{pq} 1 \quad &\Rightarrow pq | x^2 - 1 \quad &\Rightarrow pq | (x+1)(x-1); \\ x \not\equiv_{pq} 1 \quad & &\Rightarrow pq \nmid (x-1); \\ x \not\equiv_{pq} -1 \quad & &\Rightarrow pq \nmid (x+1). \end{aligned}$$

La factorización prima de $(x+1)(x-1)$ contiene un factor de p y un factor de q. Pero ni $x+1$ ni $x-1$ contienen factores de *ambos* p y q. Por lo tanto, $x+1$ y $x-1$ deben contener cada uno factores de exactamente uno de $\{p, q\}$. En otras palabras, $\{\text{MCD}(pq, x-1), \text{MCD}(pq, x+1)\} = \{p, q\}$. ■

14.4.2. Encontrando raíces cuadradas de la unidad

Afirmación 14.4.4. *Si existe un algoritmo eficiente para calcular $d \equiv_{\phi(N)} e^{-1}$ dados N y e, entonces existe un algoritmo eficiente para calcular raíces cuadradas no triviales módulo N. (Este es el paso (3) $\Rightarrow$ (4) en el Teorema 14.4.1).*

Demostración. Supongamos que tenemos un algoritmo `FIND_D` que, al recibir (N, e) como entrada, devuelve el exponente d correspondiente. Entonces, consideremos el siguiente algoritmo que usa `FIND_D` como una subrutina:

SRU(N):
Elige e como un número primo aleatorio de n bits.
$d := \texttt{FIND\_D}(N, e)$
escribe $ed - 1 = 2^s r$, con r impar
// i.e., factorizar tantos $2s$ como sea posible.
$w \leftarrow \mathbb{Z}_N$
if MCD $(w, N) \neq 1$: // $w \notin \mathbb{Z}_N^*$
    Usar MCD(w, N) para facotrizar $N = pq$
    Calcular una raíz cuadrada no trivial de la unidad utilizando p y q.
$x := w^r \mod N$
if $x \equiv_N 1$ then return 1
for $i = 0$ to s:
    if $x^2 \equiv_N 1$ then return x
    $x := x^2 \mod N$

Hay varias sentencias de retorno en este algoritmo, y debería quedar claro que todas ellas, de hecho, devuelven una raíz cuadrada de la unidad. Además, el algoritmo eventualmente regresa dentro del bucle principal `for`, porque x toma la secuencia de valores:

$$w^r, w^{2r}, w^{4r}, w^{8r}, \ldots, w^{2^s r}$$

y el valor final de esa secuencia satisface

$$w^{2^s r} = w^{ed-1} \equiv_N w^{(ed-1) \mod \phi(N)} = w^{1-1} = 1.$$

Aunque no lo probemos aquí, es posible demostrar que el algoritmo devuelve una raíz cuadrada de la unidad elegida uniformemente al azar entre las cuatro posibles raíces cuadradas de la unidad. Por lo tanto, con una probabilidad de $1/2$ la salida es una raíz cuadrada no trivial. Podemos repetir este proceso básico n veces, y eventualmente encontrar una raíz cuadrada no trivial de la unidad con una probabilidad de $1 - 2^{-n}$.

■

Capítulo 15

Establecimiento de clave de Diffie-Hellman

15.1. Grupos cíclicos

Definición 15.1.1. *Sea $g \in \mathbb{Z}_n^*$. Definimos $(g)_n = \{g^i \mod n \mid i \in \mathbb{Z}\}$, el conjunto de todas las potencias de g reducidas módulo n. Entonces g se llama generador de $(g)_n$, y $(g)_n$ se llama el grupo cíclico generado por g módulo n. Si $(g)_n = \mathbb{Z}_n^*$, entonces decimos que g es una raíz primitiva módulo n.*

La definición permite que el generador g se eleve a un entero negativo. Dado que $g \in \mathbb{Z}_n^*$, se garantiza que g tiene un inverso multiplicativo módulo n, al que podemos llamar g^{-1}. Entonces g^{-i} se puede definir como $g^{-i} \stackrel{\text{def}}{=} (g^{-1})^i$. Todas las leyes habituales de los exponentes se cumplen con respecto a esta definición de exponentes negativos.

Ejemplo 15.1. Tomando $n = 13$, tenemos:

- $(1)_{13} = \{1\}$
- $(2)_{13} = \{1, 2, 4, 8, 3, 6, 12, 11, 9, 5, 10, 7\} = \mathbb{Z}_{13}^*$
- $(3)_{13} = \{1, 3, 9\}$

Por lo tanto, 2 es una raíz primitiva módulo 13. Cada uno de los grupos $\{1\}$, $\mathbb{Z}_{13}^*$, $\{1, 3, 9\}$ es un grupo cíclico bajo la multiplicación módulo 13.

Un grupo cíclico puede tener más de un generador, por ejemplo:

$(3)_{13} = (9)_{13} = \{1, 3, 9\}$

De manera similar, hay cuatro raíces primitivas módulo 13 (equivalentemente, $\mathbb{Z}_{13}^*$ tiene cuatro generadores diferentes); son 2, 6, 7 y 11.

No todo entero tiene una raíz primitiva. Por ejemplo, no hay raíz primitiva módulo 15. Sin embargo, cuando p es un primo, siempre hay una raíz primitiva módulo p (y por lo tanto $\mathbb{Z}_p^*$ es un grupo cíclico).

Escribamos $\mathbb{G} = \langle g \rangle = \{g^i \mid i \in \mathbb{Z}\}$ para denotar un grupo cíclico no especificado generado por g. La propiedad definitoria de $\mathbb{G}$ es que cada uno de sus elementos se puede escribir como una potencia de g. De esto podemos concluir que:

- Cualquier grupo cíclico es cerrado bajo la multiplicación. Es decir, tomemos cualesquiera $X, Y \in \mathbb{G}$; entonces debe ser posible escribir $X = g^x$ e $Y = g^y$ para algunos enteros x, y. Usando la operación de multiplicación de $\mathbb{G}$, el producto es $XY = g^{x+y}$, que también está en $\mathbb{G}$.
- Cualquier grupo cíclico es cerrado bajo inversos. Tomemos cualquier $X \in \mathbb{G}$; entonces debe ser posible escribir $X = g^x$ para algún entero x. Entonces podemos ver que $g^{-x} \in \mathbb{G}$ por definición, y $g^{-x}X = g^{-x+x} = g^0$ es el elemento identidad. Entonces X tiene un inverso multiplicativo (g^{-x}) en $\mathbb{G}$.

Estos hechos demuestran que $\mathbb{G}$ es, de hecho, un *grupo* en la terminología del álgebra abstracta.

15.1.1. Logaritmos discretos

Es típicamente fácil computar el valor de g^x en un grupo cíclico, dados g y x. Por ejemplo, cuando se usa un grupo cíclico de la forma $\mathbb{Z}_n^*$, podemos calcular fácilmente la exponenciación modular $g^x \mod n$ usando la exponenciación binaria (o por cuadrados repetidos).

La operación inversa en un grupo cíclico se llama el problema del logaritmo discreto:

Definición 15.1.2 (Logaritmo Discreto)**.** *El problema del logaritmo discreto es: dado $X \in \langle g \rangle$, determinar un número x tal que $g^x = X$. Aquí la exponenciación es con respecto a la operación de multiplicación en $\mathbb{G} = \langle g \rangle$.*

Se conjetura que el problema del logaritmo discreto es difícil (es decir, no existe un algoritmo de tiempo polinómico para el problema) en ciertos tipos de grupos cíclicos.

15.2. Establecimiento de claves de Diffie-Hellman

El establecimiento de claves se refiere al problema de establecer un canal privado utilizando comunicación pública. Supongamos que Alice y Bob nunca han hablado antes y no tienen secretos compartidos. Intercambiando mensajes *públicos* (es decir, que pueden ser vistos por cualquier observador externo), les gustaría establecer un secreto que sea conocido solo por ellos dos.

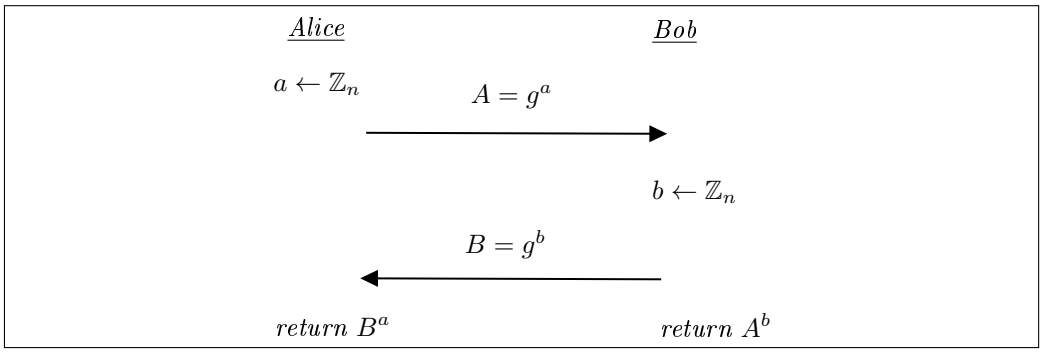
El protocolo Diffie-Hellman es un protocolo de establecimiento de claves de este tipo, y fue la primera instancia publicada de criptografía de clave pública:

Definición 15.2.1 (Diffie-Hellman)**.** *Ambas partes acuerdan (públicamente) un grupo cíclico $\mathbb{G}$ con generador g. Sea $n = |\mathbb{G}|$. Todas las exponenciaciones son con respecto a la operación del grupo en $\mathbb{G}$.*

1. *Alice elige $a \leftarrow \mathbb{Z}_n$. Ella envía $A = g^a$ a Bob.*

2. *Bob elige* $b \leftarrow \mathbb{Z}_n$. *Él envía* $B = g^b$ *a Alice.*

3. *Bob calcula localmente* $K := A^b$. *Alice calcula localmente* $K := B^a$.



Sustituyendo y aplicando las reglas estándar de los exponentes, vemos que ambas partes producen un valor común, a saber, $K = g^{ab} \in \mathbb{G}$.

15.2.1. Definiendo la seguridad para el establecimiento de claves

La ejecución de un protocolo de establecimiento de claves deja dos artefactos. Primero, tenemos la colección de mensajes que se intercambian entre las dos partes. Llamamos a esta colección un *transcript* (transcripción). Imaginamos a dos partes ejecutando un protocolo de establecimiento de claves en presencia de un *espía no autorizado*, y por lo tanto imaginamos que el transcript es público. Segundo, tenemos la *clave* que es generada por las partes, la cual es privada.

Para definir la seguridad del establecimiento de claves, nos gustaría requerir que el transcript no filtre ninguna información (útil) al espía sobre la clave. Hay algunas maneras de abordar la definición:

- Podríamos requerir que sea difícil calcular la clave dado el transcript. Sin embargo, esta resulta ser una definición bastante débil. Por ejemplo, no descarta la posibilidad de que un espía pueda adivinar la *primera mitad* de los bits de la clave.
- Podríamos requerir que la clave sea *pseudoaleatoria* dado el transcript. Esta es una mejor definición, y la que usamos. Para formalizar esta idea, definimos dos librerías. En ambas librerías, el adversario puede obtener el transcript de una ejecución del protocolo de establecimiento de claves. En una librería, el adversario obtiene la clave que resultó de la ejecución del protocolo, mientras que en la otra librería el adversario obtiene una clave totalmente no relacionada (elegida uniformemente del conjunto $\Sigma.\mathcal{K}$ de posibles claves).

Definición 15.2.2 (Seguridad KA)**.** *Sea* Σ *un protocolo de establecimiento de claves. Escribimos* $\Sigma.\mathcal{K}$ *para el espacio de claves del protocolo (es decir, el conjunto de posibles claves que produce). Escribimos* $(t, K) \leftarrow EXECPROT(\Sigma)$ *para denotar el proceso de ejecutar el protocolo entre dos partes honestas, donde* t *denota el transcript resultante, y* K *es la clave resultante. Observe que este proceso es aleatorizado y que* K *presumiblemente está correlacionado con* t.

Decimos que Σ *es seguro si* $\mathcal{L}^{\Sigma}_{ka} \approx \mathcal{L}^{\Sigma}_{ka\$}$, *donde:*

$\mathcal{L}^{\Sigma}_{ka}$

QUERY():
$(t, K) \leftarrow EXECPROT(\Sigma)$
return (t, K)

$\mathcal{L}^{\Sigma}_{ka\text{-}\$}$

QUERY():
$(t, K) \leftarrow EXECPROT(\Sigma)$
$K' \leftarrow \Sigma.\mathcal{K}$
return (t, K')

15.3. Problema de Diffie-Hellman decisional

El protocolo Diffie-Hellman está parametrizado por la elección del grupo cíclico $\mathbb{G}$ (y el generador g). Los transcripts en el protocolo consisten en (g^a, g^b), donde a y b se eligen uniformemente. La clave correspondiente a tal transcript es g^{ab}. El conjunto de posibles claves es el grupo cíclico $\mathbb{G}$.

Sustituyamos los detalles del protocolo Diffie-Hellman en las librerías de seguridad KA. Después de simplificar, vemos que la seguridad del protocolo Diffie-Hellman es equivalente a la siguiente afirmación:

$\mathcal{L}^{\mathbb{G}}_{\text{dh}}$

QUERY():
$a, b \leftarrow \mathbb{Z}_n$
return (g^a, g^b, g^{ab})

$\approx$

$\mathcal{L}^{\mathbb{G}}_{\text{dh-\$}}$

QUERY():
$a, b, c \leftarrow \mathbb{Z}_n$
return (g^a, g^b, g^c)

Hemos renombrado las librerías a $\mathcal{L}_{\text{dh}}$ y $\mathcal{L}_{\text{dh-\$}}$. En $\mathcal{L}_{\text{dh}}$ la respuesta a QUERY corresponde a un transcript DHKA (g^a, g^b) junto con la clave "correcta" correspondiente g^{ab}. La respuesta en $\mathcal{L}_{\text{dh-\$}}$ corresponde a un transcript DHKA junto con una clave aleatoria completamente independiente g^c.

Definición 15.3.1 (DDH)**.** *La suposición de Diffie-Hellman Decisional (DDH) en un grupo cíclico $\mathbb{G}$ es que $\mathcal{L}^{\mathbb{G}}_{dh} \approx \mathcal{L}^{\mathbb{G}}_{dh\text{-}\$}$ (librerías definidas anteriormente).*

Dado que hemos definido la suposición DDH simplemente renombrando la definición de seguridad para DHKA, tenemos inmediatamente:

Afirmación 15.3.2. *El protocolo DHKA es un protocolo KA seguro si y solo si la suposición DDH es verdadera para la elección de $\mathbb{G}$ utilizada en el protocolo.*

15.3.1. ¿Para qué grupos se cumple la suposición DDH?

Hasta ahora, nuestro único ejemplo de un grupo cíclico es $\mathbb{Z}^*_p$, donde p es un primo. Aunque muchos libros de texto describen DHKA en términos de este grupo cíclico, no es una buena elección porque la suposición DDH es demostrablemente falsa en $\mathbb{Z}^*_p$. Para ver por qué, introducimos un nuevo concepto:

Afirmación 15.3.3 (Criterio de Euler)**.** *Si p es un primo y $X = g^x \in \mathbb{Z}^*_p$, entonces $X^{\frac{p-1}{2}} \equiv_p (-1)^x$.*

Observe que $(-1)^x$ es 1 si x es par y -1 si x es impar. Entonces, mientras que en general es difícil determinar x dado g^x, el criterio de Euler dice que es posible determinar la *paridad* de x (es decir, si x es par o impar) dado g^x.

Para ver cómo estas observaciones conducen a un ataque contra el protocolo Diffie-Hellman, considere el siguiente ataque:

$\mathcal{A}$

$(A, B, C) \leftarrow \text{QUERY}()$

return $1 \stackrel{?}{\equiv}_p C^{\frac{p-1}{2}}$

A grandes rasgos, el adversario devuelve verdadero siempre que C pueda escribirse como g elevado a un exponente *par*. Cuando se vincula a $\mathcal{L}_{\text{dh}}$, $C = g^{ab}$ donde a y b se eligen uniformemente. Por lo tanto, ab será par con una probabilidad de $3/4$. Cuando se vincula a $\mathcal{L}_{\text{dh-\$}}$, $C = g^c$ para una c aleatoria independiente. Entonces, c es par solo con una probabilidad de $1/2$. Por lo tanto, el adversario distingue las librería con una ventaja de $1/4$.

Concretamente, con esta elección de grupo, la clave g^{ab} nunca se distribuirá uniformemente.

Residuos cuadráticos. Se han propuesto varias mejores opciones de grupos cíclicos en la literatura. Podría decirse que la más simple se basa en la siguiente definición:

Definición 15.3.4. *Un número $X \in \mathbb{Z}_n^*$ es un residuo cuadrático módulo n si existe algún entero Y tal que $Y^2 \equiv_n X$. Es decir, si X se puede obtener elevando al cuadrado un número módulo n. Sea $\mathbb{QR}_n^* \subseteq \mathbb{Z}_n^*$ el conjunto de residuos cuadráticos módulo n.*

Para nuestros propósitos, es suficiente saber que, cuando p es primo, $\mathbb{QR}_p^*$ es un grupo cíclico con $(p-1)/2$ elementos. Cuando tanto p como $(p-1)/2$ son primos, llamamos a p un *primo seguro* (y llamamos a $(p-1)/2$ un *primo de Sophie Germain*). Hasta donde sabemos, la suposición DDH es verdadera en $\mathbb{QR}_p^*$ cuando p es un primo seguro.

Capítulo 16

Cifrado de clave pública

Hasta ahora, los esquemas de cifrado que hemos visto son esquemas de *clave simétrica*. Se usa la misma clave para cifrar y descifrar. En este capítulo introducimos esquemas de cifrado de *clave pública* (a veces llamados asimétricos), que usan diferentes claves para el cifrado y el descifrado. La idea es que la clave de cifrado se puede hacer pública, de modo que cualquiera pueda enviar un cifrado al propietario de esa clave, incluso si los dos usuarios nunca antes han hablado y no comparten secretos. La clave de descifrado es privada, de modo que solo el propietario designado puede descifrar.

Modificamos la sintaxis de un esquema de cifrado de la siguiente manera. Un esquema de cifrado de clave pública consta de los siguientes tres algoritmos:

- **Gen:** Produce un par (pk, sk) donde pk es una clave pública y sk es una clave privada/secreta.
- **Enc:** Toma la clave pública pk y un texto plano m como entrada, y produce un texto cifrado c.
- **Dec:** Toma la clave secreta sk y un texto cifrado c como entrada, y produce un texto plano m.

Modificamos la condición de corrección de manera similar. Un esquema de cifrado de clave pública satisface la corrección si, para todo $m \in \mathcal{M}$ y todo $(pk, sk) \leftarrow \text{Gen}$, tenemos $\text{Dec}(sk, \text{Enc}(pk, m)) = m$ (con probabilidad 1 sobre la aleatoriedad de Enc).

16.1. Definiciones de seguridad

Ahora modificamos la definición de seguridad CPA para que se ajuste al contexto del cifrado de clave pública. Como antes, el adversario llama a una subrutina CHALLENGE con dos textos planos; la diferencia entre las dos librerías es qué texto plano se cifra realmente. Por supuesto, la operación de cifrado ahora toma la clave pública.

Entonces, el mayor cambio es que nos gustaría hacer pública la clave pública. En otras palabras, el adversario debería tener una forma de conocer la clave pública (de lo contrario, la librería no puede modelar una situación en la que el adversario conoce la clave pública). Para hacer esto, simplemente agregamos otra subrutina que devuelve la clave pública.

Definición 16.1.1 (Seguridad CPA de clave pública)**.** *Sea Σ un esquema de cifrado. Decimos que Σ tiene seguridad contra ataques de texto plano elegido (seguridad pk-CPA) si $\mathcal{L}^{\Sigma}_{\text{pk-cpa-0}} \approx \mathcal{L}^{\Sigma}_{\text{pk-cpa-1}}$ donde:*

$\mathcal{L}^{\Sigma}_{\text{pk-cpa-0}}$

$(pk, sk) \leftarrow \Sigma.Gen$

$\underline{GETPK():}$
$return\ pk$

$\underline{CHALLENGE(m_0, m_1 \in \Sigma.\mathcal{M}):}$
$return\ \ \Sigma.Enc(pk, m_0)$

$\mathcal{L}^{\Sigma}_{\text{pk-cpa-1}}$

$(pk, sk) \leftarrow \Sigma.Gen$

$\underline{GETPK():}$
$return\ pk$

$\underline{CHALLENGE(m_0, m_1 \in \Sigma.\mathcal{M}):}$
$return\ \ \Sigma.Enc(pk, m_1)$

Al igual que en el caso de los cifrados de clave secreta, los crifados deterministas no proporcionan seguridad CPA. Llamar a $\text{Enc}(k, m)$ dos veces, con la misma clave pública y el mismo texto plano, conduce al mismo texto cifrado. Recuerda que uno de los primeros y más importantes aspectos de la seguridad CPA es que es incompatible con el cifrado determinista. **¡El cifrado determinista nunca puede ser seguro bajo CPA!**

16.1.1. Textos cifrados pseudoaleatorios

Podemos modificar/adaptar la definición de textos cifrados pseudoaleatorios al cifrado de clave pública de manera similar:

Definición 16.1.2 (Seguriad pk-pCPA)**.** *Sea Σ un esquema de cifrado. Decimos que Σ tiene* ***textos cifrados pseudoaleatorios en presencia de ataques de texto plano elegido (seguridad pk-pCPA)*** *si $\mathcal{L}^{\Sigma}_{\text{pk-pCPA}} \approx \mathcal{L}^{\Sigma}_{\text{pk-pCPA-\$}}$, donde:*

$\mathcal{L}^{\Sigma}_{\text{pk-pCPA}}$

$(pk, sk) \leftarrow \Sigma.Gen$

$\underline{GETPK():}$
$return\ pk$

$\underline{CHALLENGE(m \in \Sigma.\mathcal{M}):}$
$return\ \ \Sigma.Enc(pk, m)$

$\mathcal{L}^{\Sigma}_{\text{pk-pCPA-\$}}$

$(pk, sk) \leftarrow \Sigma.Gen$

$\underline{GETPK():}$
$return\ pk$

$\underline{CHALLENGE(m \in \Sigma.\mathcal{M}):}$
$c \leftarrow \Sigma.\mathcal{C}$
$return\ \ c$

Como en el contexto de clave simétrica, la seguridad pk-pCPA (para el cifrado de clave pública) implica la seguridad CPA:

Afirmación 16.1.3. *Sea Σ un esquema de cifrado de clave pública. Si Σ tiene seguridad pk-pCPA, entonces Σ tiene seguridad CPA.*

La prueba es extremadamente similar a la prueba de la afirmación análoga para el cifrado de clave simétrica (Afirmación 9.2.2), y se deja como un ejercicio.

16.2. La seguridad de un solo uso implica la seguridad de múltiples usos

Hasta ahora, todo sobre el cifrado de clave pública ha sido directamente análogo a lo que hemos visto sobre el cifrado de clave simétrica. Ahora discutimos una propiedad peculiar que es diferente entre los dos contextos.

En el cifrado de clave simétrica, vimos ejemplos de esquemas de cifrado que son seguros cuando el adversario ve solo un texto cifrado, pero inseguros cuando el adversario ve más textos cifrados. El one-time pad es el ejemplo estándar de dicho esquema de cifrado.

Sorprendentemente, si un esquema de cifrado de clave pública es seguro cuando el adversario ve solo un texto cifrado, ¡entonces también es seguro para muchos textos cifrados! En resumen, no existe un one-time pad de clave pública que sea más débil que el cifrado de clave pública completo; hay cifrado de clave pública o nada.

Para mostrar esta propiedad formalmente, primero adaptamos la definición de secreto de un solo uso (Definición 5.4.1) al contexto de clave pública. Hay una sutileza técnica pequeña pero importante: en la Definición 5.4.1, la clave de cifrado se elige en el último momento posible en el cuerpo de CHALLENGE. Esto asegura que la clave sea local a este ámbito y, por lo tanto, cada valor de la clave solo se usa para cifrar un texto plano.

Sin embargo, en el contexto de clave pública, resulta importante permitir que el adversario vea la clave pública antes de decidir qué textos planos cifrar. (Esta preocupación no está presente en el contexto de clave simétrica precisamente porque no hay nada público de lo que pueda depender la elección de textos planos del adversario). Por esa razón, en el contexto de clave pública debemos muestrear las claves en el momento de la inicialización para que el adversario pueda obtener la clave pública a través de GETPK. Para asegurar que la clave se use para cifrar solo un texto plano, agregamos un contador y una condición de guarda a CHALLENGE, de modo que solo responda una vez con un texto cifrado.

Definición 16.2.1 (Seguridad pk-ots)**.** *Sea Σ un esquema de cifrado de clave pública. Entonces Σ tiene secreto de un solo uso si $\mathcal{L}^{\Sigma}_{\text{pk-ots-0}} \approx \mathcal{L}^{\Sigma}_{\text{pk-ots-1}}$, donde:*

$\mathcal{L}^{\Sigma}_{pk\text{-}ots\text{-}0}$

$(pk, sk) \leftarrow \Sigma.Gen$
count:=0

GETPK():
return pk

CHALLENGE$(m_0, m_1 \in \Sigma.\mathcal{M})$*:*
count:=count+1
if count >1: return null
return $\Sigma.Enc(pk, m_0)$

$\mathcal{L}^{\Sigma}_{pk\text{-}ots\text{-}1}$

$(pk, sk) \leftarrow \Sigma.Gen$
count:=0

GETPK():
return pk

CHALLENGE$(m_0, m_1 \in \Sigma.\mathcal{M})$*:*
count:=count+1
if count >1: return null
return $\Sigma.Enc(pk, m_1)$

Afirmación 16.2.2. *Sea Σ un esquema de cifrado de clave pública. Si Σ tiene secreto de un solo uso, entonces Σ es seguro bajo CPA.*

Demostración. Supongamos que $\mathcal{L}^{\Sigma}_{\text{pk-ots-0}} \approx \mathcal{L}^{\Sigma}_{\text{pk-ots-1}}$. Nuestro objetivo es mostrar que $\mathcal{L}^{\Sigma}_{\text{pk-cpa-0}} \approx \mathcal{L}^{\Sigma}_{\text{pk-cpa-1}}$. La demostración se centra en la siguiente librería $\mathcal{L}^{\Sigma}_{\text{hyb-h}}$, que está diseñada para estar vinculada a $\mathcal{L}^{\Sigma}_{\text{pk-ots-0}}$ o $\mathcal{L}_{\text{pk-ots-1}}$:

$\mathcal{L}^{\Sigma}_{\text{hyb-h}}$

count:=0
pk:=GETPK()

CHALLENGE$(m_0, m_1 \in \Sigma.\mathcal{M})$:
count:=count+1
if count $<$ h
    return Σ.Enc(pk, m_1)
else if count$=$ h
    return CHALLENGE$'(m_0, m_1)$
else
    return Σ.Enc(pk, m_0)

Aquí, el valor h es un valor no especificado que será una constante codificada, y CHALLENGE' (llamado por la rama "else if") y GETPK se refieren a la subrutina en $\mathcal{L}_{\text{pk-ots-0}}$ y $\mathcal{L}_{\text{pk-ots-1}}$. Observe que $\mathcal{L}_{\text{hyb-h}}$ está diseñado para que solo realice una llamada a CHALLENGE'; en particular, solo cuando su propia subrutina CHALLENGE se llama por h-ésima vez.

Ahora hacemos algunas observaciones:

$\mathcal{L}_{\text{pk-cpa-0}} \equiv \mathcal{L}_{\text{hyb-1}} \diamond \mathcal{L}_{\text{pk-ots-0}}$: En ambas librerías, cada llamada a CHALLENGE cifra el texto plano iz

$\mathcal{L}_{\text{hyb-h}} \diamond \mathcal{L}_{\text{pk-ots-0}} \approx \mathcal{L}_{\text{hyb-h}} \diamond \mathcal{L}_{\text{pk-ots-1}}$,

$\mathcal{L}_{\text{hyb-h}} \diamond \mathcal{L}_{\text{pk-ots-1}} = \mathcal{L}_{\text{hyb-(h+1)}} \diamond \mathcal{L}_{\text{pk-ots-0}}$,

$\mathcal{L}_{\text{hyb-q}} \diamond \mathcal{L}_{\text{pk-ots-1}} = \mathcal{L}_{\text{pk-cpa-1}}$,

Reuniendo todo, tenemos que:

$$\begin{aligned}\mathcal{L}_{\text{pk-cpa-0}} &= \mathcal{L}_{\text{hyb-1}} \diamond \mathcal{L}_{\text{pk-ots-0}} \approx \mathcal{L}_{\text{hyb-1}} \diamond \mathcal{L}_{\text{pk-ots-1}} \\ &= \mathcal{L}_{\text{hyb-2}} \diamond \mathcal{L}_{\text{pk-ots-0}} \approx \mathcal{L}_{\text{hyb-2}} \diamond \mathcal{L}_{pk-ots-1} \\ &\vdots \\ &= \mathcal{L}_{\text{hyb-q}} \diamond \mathcal{L}_{\text{pk-ots-0}} \approx \mathcal{L}_{\text{hyb-q}} \diamond \mathcal{L}_{\text{pk-ots-1}} \\ &= \mathcal{L}_{\text{pk-cpa-1}},\end{aligned}$$

y por lo tanto $\mathcal{L}_{\text{pk-cpa-0}} \approx \mathcal{L}_{\text{pk-cpa-1}}$.

■

La razón por la que esta demostración funciona para el cifrado de clave pública pero no para el cifrado de clave simétrica es que *cualquiera puede cifrar* en un esquema de clave pública. En un esquema de clave simétrica, no es posible generar cifrados sin la clave. Pero en un esquema de clave pública, la clave de cifrado es pública.

Con más detalle, la librería $\mathcal{L}_{\text{hyb-h}}$ puede, de hecho, obtener pk de $\mathcal{L}_{\text{pk-ots-0}}$ y $\mathcal{L}_{\text{pk-ots-1}}$. Por lo tanto, tiene suficiente información para realizar los cifrados para todas las llamadas a CHALLENGE. De hecho, se puede pensar en $\mathcal{L}_{\text{hyb-0}}$ como haciendo todo lo que hace $\mathcal{L}_{\text{pk-cpa-0}}$, aunque no conozca la clave secreta. Dejamos que $\mathcal{L}_{\text{hyb-h}}$ designe la h-ésima llamada a CHALLENGE como una especial para ser manejada por $\mathcal{L}_{\text{pk-ots-0}}$ y $\mathcal{L}_{\text{pk-ots-1}}$. Esto nos permite cambiar el h-ésimo cifrado de usar m_0 a m_1.

16.3. Cifrado ElGamal

El cifrado ElGamal es un esquema de cifrado de clave pública que se basa en DHKA.

Definición 16.3.1 (Cifrado ElGamal)**.** *Los parámetros públicos son una elección de un grupo cíclico $\mathbb{G}$ con n elementos y un generador g.*

	Gen:	*Enc($A, M \in \mathbb{G}$):*	*Dec($a, (B, X)$):*
$\mathcal{M} = \mathbb{G}$	$sk := a \leftarrow \mathbb{Z}_n$	$b \leftarrow \mathbb{Z}_n$	*return* $X(B^a)^{-1}$
$\mathcal{C} = \mathbb{G}^2$	$pk := A := g^a$	$B := g^b$	
	return (pk, sk)	*return* $(B, M \cdot A^b)$	

El esquema satisface la corrección, ya que para todo M:

$$\begin{aligned}\text{Dec}(sk, \text{Enc}(pk, M)) &= \text{Dec}(sk, (g^b, M \cdot A^b)) \\ &= (M \cdot A^b)((g^b)^a)^{-1} \\ &= M \cdot (g^{ab})(g^{ab})^{-1} \\ &= M.\end{aligned}$$

16.3.1. Seguridad

Imagina un adversario que está interesado en atacar un esquema ElGamal. Este adversario ve la clave pública $A(= g^a)$ y un texto cifrado $(B, X)(= (g^b, Mg^{ab}))$. Intuitivamente, la suposición de Diffie-Hellman Decisional dice que el valor $X(= Mg^{ab})$ parece aleatorio, incluso para alguien que ha visto $A(= g^a)$ y $B(= g^b)$. Por lo tanto, el mensaje M se enmascara con un elemento de grupo pseudoaleatorio; como hemos visto antes, esto se parece mucho a enmascarar el mensaje con un pad aleatorio como en el one-time pad. El único cambio aquí es que, en lugar de la operación XOR, estamos usando la operación de grupo en $\mathbb{G}$.

Más formalmente, podemos probar la seguridad de ElGamal bajo la suposición DDH:

Afirmación 16.3.2. *Si la suposición DDH en el grupo $\mathbb{G}$ es verdadera, entonces ElGamal en el grupo $\mathbb{G}$ es seguro bajo pk-pCPA.*

Demostración. Es suficiente mostrar que ElGamal tiene textos cifrados pseudoaleatorios cuando el adversario solo ve un único texto cifrado. En otras palabras, mostraremos que $\mathcal{L}_{\text{pk-ots\$-real}} \approx \mathcal{L}_{\text{pk-ots\$-rand}}$, donde estas librerías son las librerías $\mathcal{L}_{\text{pk-pCPA}}$ y $\mathcal{L}_{\text{pk-pCPA-\$}}$ de la Definición 16.1.2 pero con la restricción de un único texto cifrado utilizada en la Definición 16.2.1. Se deja como ejercicio mostrar que $\mathcal{L}_{\text{pk-ots\$-real}} \approx \mathcal{L}_{\text{pk-ots\$-rand}}$ implica seguridad pk-pCPA (que a su vez implica seguridad CPA); la demostración es muy similar a la de la Afirmación 16.2.2.

La secuencia de librerías se da a continuación:

$\mathcal{L}_{\text{pk-ots\$-real}}$

$a \leftarrow \mathbb{Z}_n$
$A := g^a$
count=0

GETPK():
return A

QUERY($M \in \mathbb{G}$):
count:=count+1
if count >1: return null
$b \leftarrow \mathbb{Z}_n$
$B := g^b$
$X := M \cdot A^b$
return (B, X)

El punto de partida es la librería $\mathcal{L}_{\text{pk-ots\$-real}}$, que se muestra aquí con los detalles de ElGamal incorporados.

$a \leftarrow \mathbb{Z}_n$; $b \leftarrow \mathbb{Z}_n$
$A := g^a$; $B := g^b$; $C = A^b$
count=0

GETPK():
return A

QUERY($M \in \mathbb{G}$):
count:=count+1
if count >1: return null
$X := M \cdot C$
return (B, X)

El cuerpo principal de QUERY calcula algunos valores intermedios B y A^b. Pero dado que esas líneas solo se alcanzan una vez, no cambia nada precalcularlas en el momento de la inicialización.

$(A, B, C) \leftarrow$DHQUERY()
count=0

GETPK():
return A

QUERY($M \in \mathbb{G}$):
count:=count+1
if count >1: return null
$X := M \cdot C$
return (B, X)

$\diamond$

$\mathcal{L}^{\mathbb{G}}_{\text{dh}}$
DHQUERY():
$a, b \leftarrow \mathbb{Z}_n$
return (g^a, g^b, g^{ab})

Podemos factorizar la generación de A, B, C en términos de la librería $\mathcal{L}_{\text{dh}}$ de la definición de seguridad de Diffie-Hellman Decisional (Definición 15.3.1).

$(A, B, C) \leftarrow$DHQUERY()
count=0

GETPK():
return A

QUERY($M \in \mathbb{G}$):
count:=count+1
if count >1: return null
$X := M \cdot C$
return (B, X)

$\diamond$

$\mathcal{L}^{\mathbb{G}}_{\text{dh-\$}}$
DHQUERY():
$a, b, c \leftarrow \mathbb{Z}_n$
return (g^a, g^b, g^c)

Aplicando la seguridad de DDH, podemos reemplazar $\mathcal{L}_{\text{dh}}$ con $\mathcal{L}_{\text{dh-\$}}$.

$a, b, c \leftarrow \mathbb{Z}_n$
$A := g^a$; $B := g^b$; $C = g^c$
count=0

GETPK():
return A

QUERY($M \in \mathbb{G}$):
count:=count+1
if count >1: return null
$X := M \cdot C$
return (B, X)

La llamada a DHQUERY se ha inlineado.

$a \leftarrow \mathbb{Z}_n$
$A := g^a$
count=0

GETPK():
return A

QUERY($M \in \mathbb{G}$):
count:=count+1
if count >1: return null
$b, c \leftarrow \mathbb{Z}_n$
$B := g^b$; $C := g^c$
$X := M \cdot C$
return (B, X)

Como antes, dado que el cuerpo principal de QUERY solo se alcanza una vez, podemos mover la elección de B y C a esa subrutina en lugar de al momento de la inicialización.

$\mathcal{L}_{\text{pk-ots\$-rand}}$

$a \leftarrow \mathbb{Z}_n$
$A := g^a$
count=0

GETPK():
return A

QUERY($M \in \mathbb{G}$):
count:=count+1
if count >1: return null
b, $x \leftarrow \mathbb{Z}_n$
$B := g^b$; $X := g^x$
return (B, X)

Cuando b se muestrea uniformemente de $\mathbb{Z}_n$, la expresión $B = g^b$ es un elemento de $\mathbb{G}$ distribuido uniformemente. Recuerda también que cuando C es un elemento de $\mathbb{G}$ distribuido uniformemente, entonces $M \cdot C$ está distribuido uniformemente; esto es análogo a la propiedad del one-time pad. Aplicar este cambio da la librería a la izquierda.

En la librería final, la respuesta a QUERY es un par de elementos del grupo distribuidos uniformemente (B, X). Por lo tanto, esa librería es exactamente $\mathcal{L}_{\text{pk-ots\$-rand}}$, como se deseaba.

■

16.4. Seguridad CCA y maleabilidad

Para concluir este capítulo, adaptamos las definiciones de seguridad CCA y maleabilidad al contexto de cifrados de clave pública.

Definición 16.4.1 (Seguridad pk-CCA). *Sea Σ un esquema de cifrado. Decimos que Σ tiene* ***seguridad contra ataques de texto cifrado elegido (seguridad pk-CCA)*** *si $\mathcal{L}^{\Sigma}_{pk\text{-}cca\text{-}0} \approx \mathcal{L}^{\Sigma}_{pk\text{-}cca\text{-}1}$, donde:*

$\mathcal{L}^{\Sigma}_{pk\text{-}cca\text{-}0}$

$(pk, sk) \leftarrow \Sigma.Gen$
$\mathcal{S} := \emptyset$

GETPK():
return pk

CHALLENGE($m_0, m_1 \in \Sigma.\mathcal{M}$):
If $|m_0| \neq |m_1|$ return error
$c := \Sigma.Enc(pk, m_0)$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

DECRYPT($c \in \Sigma.\mathcal{C}$):
If $c \in \mathcal{S}$ return error
return $\Sigma.Dec(sk, c)$

$\mathcal{L}^{\Sigma}_{pk\text{-}cca\text{-}1}$

$(pk, sk) \leftarrow \Sigma.Gen$
$\mathcal{S} := \emptyset$

GETPK():
return pk

CHALLENGE($m_0, m_1 \in \Sigma.\mathcal{M}$):
If $|m_0| \neq |m_1|$ return error
$c := \Sigma.Enc(pk, m_1)$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

DECRYPT($c \in \Sigma.\mathcal{C}$):
If $c \in \mathcal{S}$ return error
return $\Sigma.Dec(sk, c)$

En esta definición, el conjunto $\mathcal{S}$ realiza un seguimiento de los textos cifrados que han sido generados por la subrutina CHALLENGE. La subrutina DECRYPT se niega a descifrar estos textos cifrados, pero con gusto descifrará cualquier otro texto cifrado de la elección del adversario. Esta restricción es crucial para la seguridad CCA, ya que evita que el adversario simplemente descifre el texto cifrado de desafío para ganar trivialmente el juego.

Definición 16.4.2 (Seguridad pk-pCCA). *Sea Σ un esquema de cifrado. Decimos que Σ tiene* ***textos cifrados pseudoaleatorios en presencia de ataques de texto cifrado elegido (seguridad pk-pCCA)*** *si $\mathcal{L}^{\Sigma}_{pk\text{-}pCCA} \approx \mathcal{L}^{\Sigma}_{pk\text{-}pCCA\text{-}\$}$ donde:*

$\mathcal{L}^{\Sigma}_{pk\text{-}pCCA}$

$(pk, sk) \leftarrow \Sigma.Gen$
$\mathcal{S} := \emptyset$

GETPK():
return pk

CHALLENGE$(m \in \Sigma.\mathcal{M})$*:*
$c := \Sigma.Enc(pk, m)$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

DECRYPT$(c \in \Sigma.\mathcal{C})$*:*
If $c \in \mathcal{S}$ *return error*
return $\Sigma.Dec(sk, c)$

$\mathcal{L}^{\Sigma}_{pk\text{-}pCCA\text{-}\$}$

$(pk, sk) \leftarrow \Sigma.Gen$
$\mathcal{S} := \emptyset$

GETPK():
return pk

CHALLENGE$(m \in \Sigma.\mathcal{M})$*:*
$c := \Sigma.\mathcal{C}(|m|)$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

DECRYPT$(c \in \Sigma.\mathcal{C})$*:*
If $c \in \mathcal{S}$ *return error*
return $\Sigma.Dec(sk, c)$

Al igual que para la seguridad pk-CPA, si un esquema tiene seguridad pk-pCCA, entonces también tiene seguridad pk-CCA, pero no viceversa.

Definición 16.4.3 (Propiedad pk-mCPA). *Sea Σ un esquema de cifrado. Decimos que Σ es* ***maleable frente a ataques con texto plano elegido (propiedad pk-mCPA)*** *si existe un adversario $\mathcal{A}$ tal que la cantidad $\Pr[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{pk\text{-}mCPA} \Rightarrow \texttt{true}]$ no es despreciable, donde*

$\mathcal{L}^{\Sigma}_{mCPA}$

$(pk, sk) := \Sigma.Gen$
$B := \emptyset$

GETPK():
return pk

MCTXT$(\ell; m_1, \ldots, m_\ell \in \Sigma.\mathcal{M})$*:*
for $i = 1$ *to* ℓ
  $B := B \cup \{m_i\}$
  $c_i := \Sigma.Enc(pk, m_i)$
return $c_1, \ldots, c_\ell$

VERIFY$(c', \mathcal{R})$*:*
$m' := \Sigma.Dec(sk, c')$
If $m' \notin \Sigma.\mathcal{M}$
  return error
return $\mathcal{R}(B, m') \stackrel{?}{=} 1$

Capítulo 17

Compartición de secretos

DNS (Domain Name System) es el sistema que mapea dominios de Internet memorables para humanos como sede.agenciatributaria.gob.es a direcciones IP legibles por máquinas como 166.123.218.220. Si un atacante puede hacerse pasar por el sistema DNS y convencer a tu ordenador que sede.agenciatributaria.gob.es en realidad reside en otra dirección IP, podría resultar en un mal día para ti.

Para protegerse contra este tipo de ataques, se ha propuesto un reemplazo llamado DNSSEC. DNSSEC utiliza criptografía para hacer imposible falsificar un mapeo de nombre de dominio. La criptografía requerida para autenticar los mapeos de DNS es ciertamente interesante, pero queda una pregunta aún más fundamental: *¿En quién se puede confiar las claves criptográficas maestras del sistema?* La organización sin fines de lucro a cargo de este tipo de cosas (ICANN) ha elegido el siguiente sistema. La clave maestra se divide en 7 partes y se distribuye en tarjetas inteligentes a 7 personas geográficamente diversas, que las guardan en cajas de seguridad.

> Al menos cinco miembros poseedores de claves de esta comunidad tendrían que reunirse en un centro de datos seguro en los Estados Unidos para reiniciar [DNSSEC] en caso de un colapso del sistema muy improbable.
>
> "Si reúnes a cinco de estos tipos, pueden descifrar [la clave raíz] en caso de que la Costa Oeste se hunda en el agua y la Costa Este sea alcanzada por una bomba nuclear", [dijo] Richard Lamb, gerente de programa para DNSSEC en ICANN.[1]

¿Cómo es posible que cualesquiera 5 de los 7 poseedores de claves puedan reconstruir la clave maestra, pero (presumiblemente) 4 de los 7 no puedan? La solución se encuentra en una herramienta criptográfica llamada **esquema de compartición de secretos**, el tema de este capítulo.

[1] http://www.livescience.com/6791-internet-key-holders-insurance-cyber-attack.html

17.1. Definiciones

Comenzamos introduciendo la sintaxis de un esquema de compartición de secretos:

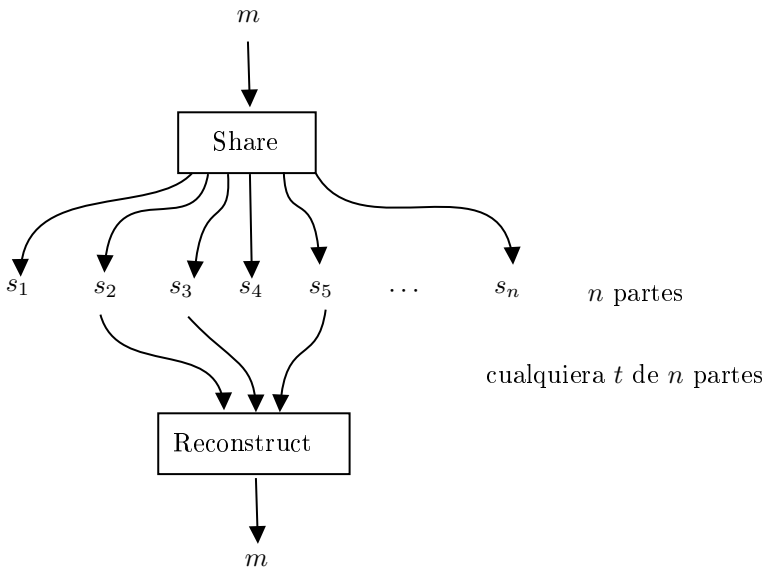
Definición 17.1.1. *Un **esquema de compartición de secretos con umbral** t **de** n **(TSSS)** consiste en los siguientes algoritmos:*

- ***Share:** un algoritmo aleatorizado que toma un **mensaje** $m \in \mathcal{M}$ como entrada y produce una secuencia $s = (s_1, \ldots, s_n)$ de **partes (shares)**.*
- ***Reconstruct:** un algoritmo determinista que toma una colección de t o más partes como entrada y produce un mensaje.*

*Llamamos $\mathcal{M}$ al **espacio de mensajes** del esquema, y t a su **umbral**. Como es habitual, nos referimos a los parámetros/componentes de un esquema Σ como $\Sigma.t$, $\Sigma.n$, $\Sigma.M$, $\Sigma.Share$, $\Sigma.Reconstruct$.*

En la compartición de secretos, numeramos a los usuarios como $\{1, \ldots, n\}$, donde el usuario i recibe la parte s_i. Sea $U \subseteq \{1, \ldots, n\}$ un subconjunto de usuarios. Entonces $\{s_i \mid i \in U\}$ se refiere al conjunto de partes que pertenecen a los usuarios U. Si $|U| \geq t$, decimos que U está **autorizado**; de lo contrario, está **no autorizado**. El objetivo de la compartición de secretos es que todos los conjuntos autorizados de usuarios/partes puedan reconstruir el secreto, mientras que todos los conjuntos no autorizados no aprenden nada.

Definición 17.1.2 (Corrección del TSSS)**.** *Un **TSSS con umbral** t **de** n satisface la **corrección** si, para todos los conjuntos autorizados $U \subseteq \{1, \ldots, n\}$ (es decir, $|U| \geq t$) y para todas las $s \leftarrow Share(m)$, tenemos $Reconstruct(\{s_i \mid i \in U\}) = m$.*



17.2. Definición de seguridad

Nos gustaría una garantía de seguridad que diga algo como:

si solo conoces un conjunto no autorizado de shares, entonces no aprendes ninguna información sobre la elección del mensaje secreto.

Para traducir esta afirmación informal a una definición de seguridad formal, definimos dos librerías que permiten al adversario aprender un conjunto de shares (para un conjunto *no autorizado*), y que difieren solo en qué secreto se comparte. Si las dos librerías son indistinguibles, entonces concluimos que ver un conjunto no autorizado de shares no revela ninguna información sobre la elección del mensaje secreto. La definición se ve así:

Definición 17.2.1 (Seguridad del TSSS)**.** *Sea Σ un esquema de compartición de secretos con umbral. Decimos que Σ es seguro si $\mathcal{L}^{\Sigma}_{tsss\text{-}0} \approx \mathcal{L}^{\Sigma}_{tsss\text{-}1}$, donde:*

$\mathcal{L}^{\Sigma}_{tsss\text{-}0}$

SHARE$(m_0, m_1 \in \Sigma.\mathcal{M}, U)$*:*
if $|U| \geq \Sigma.t$*: return err*
$s \leftarrow \Sigma.Share(m_0)$
return $\{s_i \mid i \in U\}$

$\mathcal{L}^{\Sigma}_{tsss\text{-}1}$

SHARE$(m_0, m_1 \in \Sigma.\mathcal{M}, U)$*:*
if $|U| \geq \Sigma.t$*: return err*
$s \leftarrow \Sigma.Share(m_1)$
return $\{s_i \mid i \in U\}$

En un intento de mantener la notación despejada, no hemos escrito el tipo del argumento U, que es $U \subseteq \{1, \ldots, \Sigma.n\}$.

Discusión y errores comunes

- De forma similar a la definición de secreto de un solo uso del cifrado, permitimos que el atacante elija los dos mensajes secretos que se compartirán. Como antes, esto modela un escenario de ataque en el que el adversario tiene conocimiento parcial o influencia sobre el secreto m que se comparte.
- El atacante también elige el conjunto U de partes de los usuarios que se obtendrán. Las librerías imposibilitan que el atacante obtenga las partes de un conjunto *autorizado* (devolviendo `err` en ese caso). Esto **no** significa que a un usuario nunca se le permita distribuir un número autorizado de partes (esto sería extraño, ya que haría imposible cualquier reconstrucción futura). Simplemente significa que queremos una definición de seguridad que diga algo sobre un atacante que solo ve un conjunto *no autorizado* de partes, por lo que formalizamos la seguridad en términos de librerías con esta restricción.
- Consideremos un esquema de compartición de secretos con umbral 6 de 10. Con las librerías anteriores, el atacante puede recibir las partes de los usuarios $\{1, \ldots, 5\}$ (un conjunto no autorizado) en una llamada a SHARE, y luego recibir las partes de los usuarios $\{6, \ldots, 10\}$ en otra llamada. Podría parecer que el atacante puede entonces combinar estas partes para reconstruir el secreto (si el mismo mensaje se compartió en ambas llamadas). Sin embargo, este **no** es el caso porque estos dos conjuntos de partes provinieron de dos ejecuciones independientes del algoritmo Share (un algoritmo aleatorizado). No se debe esperar que las partes generadas por una llamada a Share funcionen con las partes generadas por otra llamada, incluso si ambas llamadas a Share utilizaron el mismo mensaje secreto porque cada llamada involucra elementos aleatorios independientes.

- Recuerda que en nuestro estilo de definir la seguridad usando librerías, es solo las diferencias internas entre las librerías que deben ocultarse. Cualquier cosa que sea igual entre las dos librerías no necesita ocultarse. Una cosa que es igual para las dos librerías aquí es el hecho de que producen las partes pertenecientes al mismo conjunto de usuarios U. Esta definición de seguridad no requiere que las partes oculten a qué usuario pertenecen. De hecho, se puede modificar un esquema de compartición de secretos para que la identidad de cada usuario se agregue a su parte correspondiente, y el resultado aún satisfaría la definición de seguridad anterior.
- Al igual que la definición de cifrado no aborda el problema de la distribución de claves, la definición de compartición de secretos no aborda el problema de quién debe ejecutar el algoritmo Share (si su entrada m es tan secreta que no se puede confiar a una sola persona), o cómo se deben entregar las partes a los n usuarios diferentes. Esas preocupaciones se consideran fuera del alcance del problema de la compartición de secretos. Más bien, el enfoque se centra simplemente en si es incluso posible codificar los datos de tal manera que un conjunto no autorizado de partes no proporcione información sobre el secreto, mientras que cualquier conjunto autorizado revele completamente el secreto.

17.2.1. Un enfoque inseguro

Una forma de entender la seguridad de la compartición de secretos es ver un ejemplo de un enfoque "obvio" pero inseguro para la compartición de secretos y estudiar por qué es inseguro.

Consideremos un esquema de compartición de secretos 5 de 5. Esto significa que queremos dividir un secreto en 5 partes de modo que cualesquiera 4 de las partes no revelen nada. Una forma en que podrías pensar en hacer esto es literalmente cortar el secreto en 5 partes. Por ejemplo, si el secreto tiene 500 bits, podrías darle los primeros 100 bits al usuario 1, los segundos 100 bits al usuario 2, y así sucesivamente.

TSSS Inseguro

$\mathcal{M} = \{0,1\}^{500}$	Share(m):	Reconstruct($s_1, \ldots, s_5$):
$t = 5$	dividir m como $m = s_1 \| \cdots \| s_5$,	return $s_1 \| \cdots \| s_5$
$n = 5$	donde cada $\|s_i\| = 100$	
	return $(s_1, \ldots, s_5)$	

Es cierto que el secreto se puede construir concatenando las 5 shares, y por lo tanto esta construcción satisface la propiedad de corrección. (El único conjunto autorizado es el conjunto de los 5 usuarios, por lo que escribimos Reconstruct para esperar las 5 shares.)

Sin embargo, el esquema es inseguro (como se prometió). Supongamos que tienes incluso solo 1 share. Es cierto que no conoces el secreto en su totalidad, pero la definición de seguridad (para la compartición de secretos 5 de 5) exige que una sola share no revele *nada* sobre el secreto. Por supuesto, conocer 100 bits de algo no es lo mismo que no saber nada al respecto.

Podemos aprovechar esta observación para realizar un ataque más formal al esquema, en forma de un diferenciador entre las dos librerías $\mathcal{L}_{\text{tsss-0}}$ y $\mathcal{L}_{\text{tsss-1}}$. Como caso extremo, podemos distinguir entre shares de un secreto de todo 0 y shares de un secreto de todo 1:

$\mathcal{A}$

$s_1 := \text{SHARE}(0^{500}, 1^{500}, \{1\})$
return $s_1 \stackrel{?}{=} 0^{100}$

Enlacemos este programa que llama a ambas librerías $\mathcal{L}_{\text{tsss-0}}$ y $\mathcal{L}_{\text{tsss-1}}$ y veamos qué sucede:

$\mathcal{A}$

$s_1 := \text{SHARE}(0^{500}, 1^{500}, \{1\})$
return $s_1 \stackrel{?}{=} 0^{100}$

$\diamond$

$\mathcal{L}^{\Sigma}_{\text{tsss-0}}$

SHARE(m_0, m_1, U):
if $|U| \geq t$: return err
$s \leftarrow \text{Share}(m_0)$
return $\{s_i \mid i \in U\}$

Cuando $\mathcal{A}$ está enlazado a $\mathcal{L}^{\Sigma}_{\text{tsss-0}}$, recibe una share de 0^{500}, que en sí misma será una cadena de todos ceros. En este caso, $\mathcal{A}$ produce 1 con probabilidad 1.

$\mathcal{A}$

$s_1 := \text{SHARE}(0^{500}, 1^{500}, \{1\})$
return $s_1 \stackrel{?}{=} 0^{100}$

$\diamond$

$\mathcal{L}^{\Sigma}_{\text{tsss-1}}$

SHARE(m_0, m_1, U):
if $|U| \geq t$: return err
$s \leftarrow \text{Share}(m_1)$
return $\{s_i \mid i \in U\}$

Cuando $\mathcal{A}$ está enlazado a $\mathcal{L}^{\Sigma}_{\text{tsss-1}}$, recibe una parte de 1^{500}, que será una cadena de todos unos. En este caso, $\mathcal{A}$ produce 1 con probabilidad 0.

Hemos construido un adversario que se comporta de manera muy diferente (de hecho, tan diferente como es posible) en presencia de las dos librerías. Por lo tanto, este esquema de compartición de secretos no es seguro.

Con suerte, este ejemplo demuestra uno de los principales desafíos (y cosas asombrosas) de los esquemas de compartición de secretos. Es fácil revelar información sobre el secreto gradualmente a medida que se obtienen más shares, como en este ejemplo inseguro. Sin embargo, la definición de seguridad de la compartición de secretos es que las shares deben filtrar *absolutamente nada* de información sobre el secreto, hasta que el número de shares supere el valor del umbral.

17.3. Un esquema simple 2 de 2

Lo creas o no, ¡ya hemos visto un esquema simple de compartición de secretos! De hecho, incluso podría ser mejor pensar en el one-time pad como el esquema de compartición de secretos más simple.

TSSS 2 de 2 con one-time pad

$\mathcal{M} = \{0,1\}^{\ell}$
$t = 2$
$n = 2$

Share(m):
$s_1 \leftarrow \{0,1\}^{\ell}$
$s_2 := s_1 \oplus m$
return (s_1, s_2)

Reconstruct(s_1, s_2):
return $s_1 \oplus s_2$

Dado que es un esquema 2 de 2, el único conjunto autorizado de usuarios es $\{1, 2\}$, por lo que Reconstruct está escrito para esperar ambas shares s_1 y s_2 como sus entradas. La corrección se deduce fácilmente de lo que ya hemos aprendido sobre las propiedades de XOR.

Ejemplo 17.1. Si queremos compartir la cadena $m = 1101010001$, entonces el algoritmo Share podría elegir

$$\begin{aligned} s_1 &:= 0110000011 \\ s_2 &:= s_1 \oplus m \\ &= 0110000011 \oplus 1101010001 = 1011010010. \end{aligned}$$

Entonces, el secreto se puede reconstruir aplicando XOR a las dos shares juntas, mediante:

$$s_1 \oplus s_2 = 0110000011 \oplus 1011010010 = 1101010001 = m.$$

Recuerda que este ejemplo muestra solo una posible ejecución de Share(1101010001), pero Share es un algoritmo aleatorizado y son posibles muchos otros valores de (s_1, s_2).

Teorema 17.3.1. *La TSSS 2 a 2 con one-time pad es un esquema de compartición de secretos con umbral 2 de 2* ***seguro****.*

Demostración. Sea Σ la TSSS 2 a 2 con one-time pad. Mostraremos que $\mathcal{L}^{\Sigma}_{\text{tsss-0}} \equiv \mathcal{L}^{\Sigma}_{\text{tsss-1}}$.

$\mathcal{L}^{\Sigma}_{\text{tsss-0}}$

SHARE(m_0, m_1, U):
if $|U| \geq 2$: return err
$s_1 \leftarrow \{0,1\}^{\ell}$
$s_2 := s_1 \oplus m_0$
return $\{s_i \mid i \in U\}$

Como de costumbre, el punto de partida es $\mathcal{L}^{\Sigma}_{\text{tsss-0}}$, que se muestra aquí con los detalles del esquema de compartición de secretos ya incluidos (y los tipos de los argumentos de la subrutina omitidos para reducir el desorden).

SHARE(m_0, m_1, U):
if $|U| \geq 2$: return err
if $U = \{1\}$:
  $s_1 \leftarrow \{0,1\}^{\ell}$
  $s_2 := s_1 \oplus m_0$
  return $\{s_1\}$
else if $U = \{2\}$:
  $s_1 \leftarrow \{0,1\}^{\ell}$
  $s_2 := s_1 \oplus m_0$
  return $\{s_2\}$
else return $\emptyset$

No tiene ningún efecto en el comportamiento de la librería si duplicamos el cuerpo principal de la librería en 3 ramas de una nueva sentencia if. La razón para hacer esto es que el esquema genera s_1 y s_2 de manera diferente. Esto significa que nuestra prueba eventualmente manejará los 3 conjuntos no autorizados diferentes ($\{1\}$, $\{2\}$ y $\emptyset$) de maneras fundamentalmente diferentes.

SHARE(m_0, m_1, U):
if $|U| \geq 2$: return err
if $U = \{1\}$:
  $s_1 \leftarrow \{0,1\}^\ell$
  $s_2 := s_1 \oplus m_1$
  return $\{s_1\}$
else if $U = \{2\}$:
  $s_1 \leftarrow \{0,1\}^\ell$
  $s_2 := s_1 \oplus m_0$
  return $\{s_2\}$
else return $\emptyset$

La definición de s_2 se ha cambiado en la primera rama del if. Esto no tiene ningún efecto en el comportamiento de la librería, ya que s_2 nunca se usa realmente en esta rama.

SHARE(m_0, m_1, U):
if $|U| \geq 2$: return err
if $U = \{1\}$:
  $s_1 \leftarrow \{0,1\}^\ell$
  $s_2 := s_1 \oplus m_1$
  return $\{s_1\}$
else if $U = \{2\}$:
  $s_2 \leftarrow$ EAVESDROP(m_0, m_1)
  return $\{s_2\}$
else return $\emptyset$

$\diamond$

$\mathcal{L}^{\text{OTP}}_{\text{otint-0}}$
EAVESDROP(m_0, m_1):
$k \leftarrow \{0,1\}^\ell$
$c := k \oplus m_0$
return c

Reconociendo la segunda rama de la sentencia if como un cifrado con one-time pad (de m_0 bajo la clave s_1), factorizamos la generación de s_2 en términos de la librería $\mathcal{L}^{\text{OTP}}_{\text{otint-0}}$ de la definición de intercambiabilidad de un solo uso del one-time pad. Esto no tiene ningún efecto en el comportamiento de la librería. Es importante destacar que la subrutina en $\mathcal{L}^{\text{OTP}}_{\text{otint-0}}$ espera dos argumentos, por lo que eso es lo que debemos pasar. Elegimos pasar m_0 y m_1 por razones que deberían quedar claras muy pronto.

SHARE(m_0, m_1, U):
if $|U| \geq 2$: return err
if $U = \{1\}$:
  $s_1 \leftarrow \{0,1\}^\ell$
  $s_2 := s_1 \oplus m_1$
  return $\{s_1\}$
else if $U = \{2\}$:
  $s_2 \leftarrow$ EAVESDROP(m_0, m_1)
  return $\{s_2\}$
else return $\emptyset$

$\diamond$

$\mathcal{L}^{\text{OTP}}_{\text{otint-1}}$
EAVESDROP(m_0, m_1):
$k \leftarrow \{0,1\}^\ell$
$c := k \oplus m_1$
return c

Hemos reemplazado $\mathcal{L}^{\text{OTP}}_{\text{otint-0}}$ con $\mathcal{L}^{\text{OTP}}_{\text{otint-1}}$. De la seguridad del one-time pad (y el lema de composición), este cambio no tiene ningún efecto en el comportamiento de la librería.

SHARE(m_0, m_1, U):
if $|U| \geq 2$: return err
if $U = \{1\}$:
  $s_1 \leftarrow \{0,1\}^\ell$
  $s_2 := s_1 \oplus m_0$
  return $\{s_1\}$
else if $U = \{2\}$:
  $s_1 \leftarrow \{0,1\}^\ell$
  $s_2 := s_1 \oplus m_1$
  return $\{s_2\}$
else return $\emptyset$

Se ha puesto una subrutina en línea; sin efecto en el comportamiento de la librería.

$\mathcal{L}^{\Sigma}_{\text{tsss-1}}$
SHARE(m_0, m_1, U):
if $|U| \geq 2$: return err
$s_1 \leftarrow \{0,1\}^\ell$
$s_2 := s_1 \oplus m_1$
return $\{s_i \mid i \in U\}$

El código se ha simplificado. Específicamente, las ramas de la sentencia if se pueden unificar, sin ningún efecto en el comportamiento de la librería. El resultado es $\mathcal{L}^{\Sigma}_{\text{tsss-1}}$.

Hemos mostrado que $\mathcal{L}^{\Sigma}_{\text{tsss-0}} \equiv \mathcal{L}_{\text{hyb-1}} \equiv \cdots \equiv \mathcal{L}_{\text{hyb-5}} \equiv \mathcal{L}^{\Sigma}_{\text{tsss-1}}$, y por lo tanto el esquema de compartición de secretos es seguro. ■

De hecho, hemos probado una afirmación ligeramente más general. La única propiedad del one-time pad que hemos usado ha sido su intercambiabilidad de un solo uso. Sustituir el one-time pad por cualquier otro esquema de cifrado con intercambiabilidad de un solo uso aún permitiría que la misma prueba se llevara a cabo. Así que en realidad hemos probado lo siguiente:

Teorema 17.3.2. *Si Σ es un esquema de cifrado con intercambiabilidad de un solo uso, entonces el siguiente esquema de compartición de secretos con umbral 2 de 2 $\mathcal{S}$ es seguro:*

$\mathcal{M} = \Sigma.\mathcal{M}$	*Share(m):*	*Reconstruct(s_1, s_2):*
$t = 2$	$s_1 \leftarrow \Sigma.Gen$	*return* $\Sigma.Dec(s_1, s_2)$
$n = 2$	$s_2 \leftarrow \Sigma.Enc(s_1, m)$	
	return (s_1, s_2)	

17.4. Interpolación polinómica

Probablemente estés familiarizado con el hecho de que dos puntos determinan una línea (en geometría euclidiana). También es cierto que 3 puntos determinan una parábola, y así sucesivamente. El siguiente esquema de compartición de secretos que discutiremos se basa en el siguiente principio:

$d + 1$ puntos determinan un único polinomio de grado d.

Una nota sobre la terminología: Si f es un polinomio que se puede escribir como $f(x) = \sum_{i=0}^{d} f_i x^i$, entonces decimos que f es un polinomio de **grado** d. Sería más técnicamente correcto decir que el grado de f es *a lo sumo* d ya que permitimos que el coeficiente principal f_d sea cero. Por conveniencia, seguiremos diciendo "grado d" para significar "grado a lo sumo d".

17.4.1. Polinomios sobre los Rreales

Teorema 17.4.1 (Interpolación polinómica)**.** *Sea $\{(x_1, y_1), \ldots, (x_{d+1}, y_{d+1})\} \subseteq \mathbb{R}^2$ un conjunto de puntos cuyos valores x_i son todos distintos. Entonces existe un único polinomio f de grado d con coeficientes reales que satisface $y_i = f(x_i)$ para todo i.*

Demostración. Para empezar, consideremos el siguiente polinomio:

$$l_1(x) = \frac{(x - x_2)(x - x_3)\cdots(x - x_{d+1})}{(x_1 - x_2)(x_1 - x_3)\cdots(x_1 - x_{d+1})}.$$

La notación es potencialmente confusa. El polinomio l_1 es un polinomio con variable formal x. Los valores x_1 son solo números simples (escalares), dados en el enunciado del teorema. Por lo tanto, el numerador en l_1 es un polinomio de grado d en x. El denominador es solo un escalar, y dado que todos los x_i son distintos, no estamos dividiendo por cero. En general, l_1 es un polinomio de grado d.

¿Qué sucede cuando evaluamos l_1 en uno de los valores especiales x_i?

- Evaluar $l_1(x_1)$ hace que el numerador y el denominador sean iguales, por lo que $l_1(x_1) = 1$.
- Evaluar $l_1(x_i)$ para $i \neq 1$ conduce a un término $(x_i - x_i)$ en el numerador, por lo que $l_1(x_i) = 0$.

Por supuesto, l_1 se puede evaluar en cualquier punto (no solo en los puntos especiales $x_1, \ldots, x_{d+1}$), pero no nos importa lo que sucede en esos casos.

Podemos definir de manera similar otros polinomios l_j:

$$l_j(x) = \frac{(x - x_1)\cdots(x - x_{j-1})(x - x_{j+1})\cdots(x - x_{d+1})}{(x_j - x_1)\cdots(x_j - x_{j-1})(x_j - x_{j+1})\cdots(x_j - x_{d+1})}.$$

El patrón es que al numerador le "falta" el término $(x - x_j)$ y al denominador le falta el término $(x_j - x_j)$, porque no queremos un cero en el denominador. Los polinomios de este tipo se llaman **polinomios de Lagrange**. Cada uno de ellos son polinomios de grado d, y satisfacen la propiedad:

$$l_j(x_i) = \begin{cases} 1 & \text{si } i = j \\ 0 & \text{si } i \neq j. \end{cases}$$

Ahora consideremos el siguiente polinomio:

$$f(x) = y_1 l_1(x) + y_2 l_2(x) + \cdots + y_{d+1} l_{d+1}(x).$$

Nótese que f es un polinomio de grado d ya que es la suma de polinomios de grado d (de nuevo, los valores y_i son solo escalares).

¿Qué sucede cuando evaluamos f en uno de los valores especiales x_i? Dado que $l_i(x_i) = 1$ y $l_j(x_i) = 0$ para $j \neq i$, obtenemos:

$$\begin{aligned} f(x_i) &= y_1 l_1(x_i) + \cdots + y_i l_i(x_i) + \cdots + y_{d+1} l_{d+1}(x_i) \\ &= y_1 \cdot 0 + \cdots + y_i \cdot 1 + \cdots + y_{d+1} \cdot 0 \\ &= y_i \end{aligned}$$

Así que $f(x_i) = y_i$ para cada x_i, que es lo que queríamos. Esto muestra que existe algún polinomio de grado d con esta propiedad.

Ahora argumentemos que esta f es única. Supongamos que hay dos polinomios de grado d, f y g, tales que $f(x_i) = g(x_i) = y_i$ para $i \in \{1, ..., d+1\}$. Entonces el polinomio $h(x) = f(x) - g(x)$ también es de grado d, y satisface $h(x_i) = 0$ para todo i. En otras palabras, cada x_i es una raíz de h, por lo que h tiene al menos $d+1$ raíces. Pero el único polinomio de grado d con $d+1$ raíces es el polinomio idénticamente cero $h(x) = 0$. Si $h(x) = 0$ entonces $f = g$. En otras palabras, cualquier polinomio de grado d, g, que satisfaga $g(x_i) = y_i$ debe ser igual a f. Así que f es el único polinomio con esta propiedad. ■

Ejemplo 17.2. Averigüemos el polinomio de grado 3 que pasa por los puntos (3, 1), (4, 1), (5, 9), (2, 6):

i	1	2	3	4
x_i	3	4	5	2
y_i	1	1	9	6

Primero, construyamos los polinomios de Lagrange apropiados:

$$l_1(x) = \frac{(x-x_2)(x-x_3)(x-x_4)}{(x_1-x_2)(x_1-x_3)(x_1-x_4)} = \frac{(x-4)(x-5)(x-2)}{(3-4)(3-5)(3-2)} = \frac{x^3-11x^2+38x-40}{2}$$

$$l_2(x) = \frac{(x-x_1)(x-x_3)(x-x_4)}{(x_2-x_1)(x_2-x_3)(x_2-x_4)} = \frac{(x-3)(x-5)(x-2)}{(4-3)(4-5)(4-2)} = \frac{x^3-10x^2+31x-30}{-2}$$

$$l_3(x) = \frac{(x-x_1)(x-x_2)(x-x_4)}{(x_3-x_1)(x_3-x_2)(x_3-x_4)} = \frac{(x-3)(x-4)(x-2)}{(5-3)(5-4)(5-2)} = \frac{x^3-9x^2+26x-24}{6}$$

$$l_4(x) = \frac{(x-x_1)(x-x_2)(x-x_3)}{(x_4-x_1)(x_4-x_2)(x_4-x_3)} = \frac{(x-3)(x-4)(x-5)}{(2-3)(2-4)(2-5)} = \frac{x^3-12x^2+47x-60}{-6}$$

Como comprobación de cordura, observe cómo:

$$l_1(x_1) = l_1(3) = \frac{3^3 - 11 \cdot 3^2 + 38 \cdot 3 - 40}{2} = \frac{2}{2} = 1$$

$$l_1(x_2) = l_1(4) = \frac{4^3 - 11 \cdot 4^2 + 38 \cdot 4 - 40}{2} = \frac{0}{2} = 0$$

Facilitará el siguiente paso si reescribimos todos los polinomios de Lagrange para que tengan el mismo denominador 6:

$$l_1(x) = \frac{3x^3-33x^2+114x-120}{6} \quad l_2(x) = \frac{-3x^3+30x^2-93x+90}{6}$$

$$l_3(x) = \frac{x^3-9x^2+26x-24}{6} \quad l_4(x) = \frac{-x^3+12x^2-47x+60}{6}$$

Nuestro polinomio deseado es

$$\begin{aligned} f(x) &= y_1 \cdot l_1(x) + y_2 \cdot l_2(x) + y_3 \cdot l_3(x) + y_4 \cdot l_4(x) \\ &= 1 \cdot l_1(x) + 1 \cdot l_2(x) + 9 \cdot l_3(x) + 6 \cdot l_4(x) \\ &= \frac{1}{6} \begin{pmatrix} 1 \cdot (3x^3 - 33x^2 + 114x - 120) \\ +1 \cdot (-3x^3 + 30x^2 - 93x + 90) \\ +9 \cdot (x^3 - 9x^2 + 26x - 24) \\ +6 \cdot (-x^3 + 12x^2 - 47x + 60) \end{pmatrix} \\ &= \frac{1}{6}(3x^3 - 12x^2 - 27x + 114) \\ &= \frac{x^3}{2} - 2x^2 - \frac{9x}{2} + 19 \end{aligned}$$

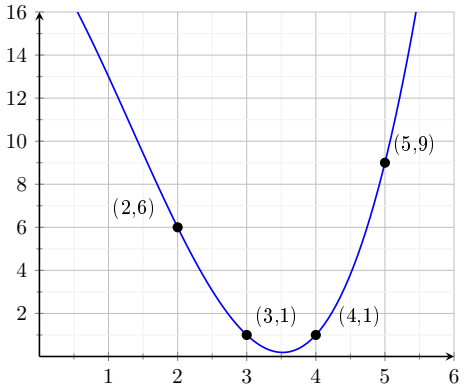
Y, de hecho, f da los valores correctos:

$$f(x_1) = f(3) = \frac{3^3}{2} - 2 \cdot 3^2 - \frac{9 \cdot 3}{2} + 19 = 1 = y_1$$

$$f(x_2) = f(4) = \frac{4^3}{2} - 2 \cdot 4^2 - \frac{9 \cdot 4}{2} + 19 = 1 = y_2$$

$$f(x_3) = f(5) = \frac{5^3}{2} - 2 \cdot 5^2 - \frac{9 \cdot 5}{2} + 19 = 9 = y_3$$

$$f(x_4) = f(2) = \frac{2^3}{2} - 2 \cdot 2^2 - \frac{9 \cdot 2}{2} + 19 = 6 = y_4.$$



17.4.2. Polinomios mod p

Veremos un esquema de compartición de secretos basado en polinomios, cuyo algoritmo Share debe elegir un polinomio con coeficientes uniformemente aleatorios. Dado que no podemos tener una distribución uniforme sobre los números reales, debemos considerar en su lugar polinomios con coeficientes en $\mathbb{Z}_p$.

¡Sigue siendo cierto que $d+1$ puntos determinan un único polinomio de grado d cuando se trabaja módulo p, si p es un *primo*!

Teorema 17.4.2 (Interpolación mód p)**.** *Sea p un primo, y sea $\{(x_1, y_1), \ldots, (x_{d+1}, y_{d+1})\} \subseteq (\mathbb{Z}_p)^2$ un conjunto de puntos cuyos valores x_i son todos distintos. Entonces existe un único polinomio f de grado d con coeficientes de $\mathbb{Z}_p$ que satisface $y_i \equiv_p f(x_i)$ para todo i.*

La demostración es la misma que la del Teorema 17.4.1, si se interpreta toda la aritmética módulo p. La suma, la resta y la multiplicación mod p son sencillas; la única cuestión no trivial es cómo interpretar la "división mod p", que es necesaria en la definición de los polinomios l_j. Pero en la Sección 14.1 vimos que siempre se puede "dividir" mod p (excepto por cero) cuando p es un primo.

También podemos generalizar la observación de que $d+1$ puntos determinan un único polinomio de grado d. Resulta que:

> Para cualquier k puntos, hay exactamente p^{d+1-k} polinomios de grado d que pasan por esos puntos, mod p.

Observa que cuando $k = d+1$, la afirmación dice que solo hay un único polinomio que pasa por los puntos.

Corolario 17.4.3 (# de polinomios)**.** *Sea $\mathcal{P} = \{(x_1, y_1), \ldots, (x_k, y_k)\} \subseteq (\mathbb{Z}_p)^2$ un conjunto de puntos cuyos valores x_i son distintos. Sea d un entero tal que $k \leq d+1$ y $p > d$. Entonces, el número de polinomios f de grado d con coeficientes en $\mathbb{Z}_p$ que satisfacen la condición $y_i \equiv_p f(x_i)$ para todo i es exactamente p^{d+1-k}.*

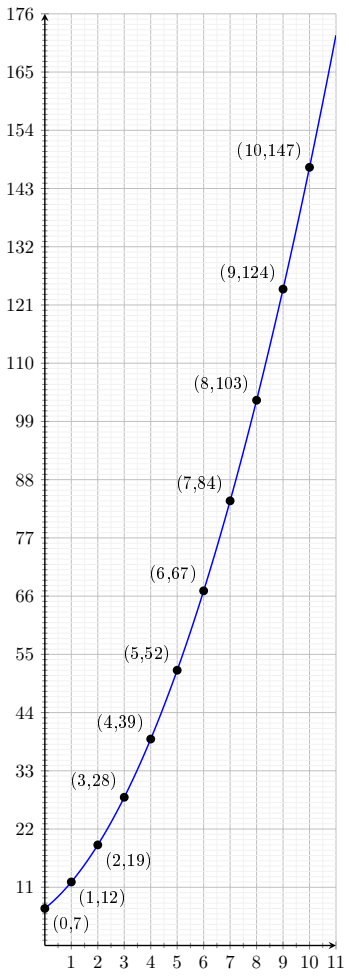
Demostración. La demostración se realiza por inducción sobre el valor $d+1-k$. El caso base es cuando $d+1-k = 0$. Entonces tenemos $k = d+1$ puntos distintos, y el Teorema 17.4.1 dice que hay un único polinomio que satisface la condición. Dado que $p^{d+1-k} = p^0 = 1$, el caso base es verdadero.

Para el caso inductivo, tenemos $k \leq d$ puntos en $\mathcal{P}$. Sea $x^* \in \mathbb{Z}_p$ un valor que no aparece como uno de los x_i. Cada polinomio debe dar algún valor cuando se evalúa en x^*. Entonces,

$$
\begin{aligned}
&[\#\text{ de polinomios de grado } d \text{ que pasan por los puntos en } \mathcal{P}] \\
&= \sum_{y^* \in \mathbb{Z}_p} [\#\text{ de polinomios de grado } d \text{ que pasan por los puntos en } \mathcal{P} \cup \{(x^*, y^*)\}] \\
&\overset{(*)}{=} \sum_{y^* \in \mathbb{Z}_p} p^{d+1-(k+1)} \\
&= p \cdot (p^{d+1-k-1}) = p^{d+1-k}
\end{aligned}
$$

La igualdad marcada con $(*)$ se deduce de la hipótesis inductiva, ya que cada uno de los términos involucra un polinomio que pasa por un conjunto especificado de $k+1$ puntos con coordenadas x distintas. ■

Ejemplo 17.3.

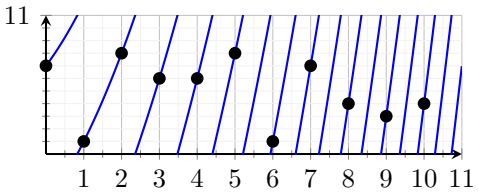


¿Cómo se ve un "polinomio módulo p"? Consideremos un ejemplo de polinomio de grado 2:

$$f(x) = x^2 + 4x + 7$$

Cuando graficamos este polinomio sobre los números reales (la imagen de la izquierda), obtenemos una parábola familiar.

Veamos cómo se "ve" este polinomio módulo 11 (es decir, en $\mathbb{Z}_{11}$). Trabajar módulo 11 significa "dar la vuelta" cada vez que el polinomio cruza un múltiplo de 11 a lo largo del eje y. Esto resulta en la gráfica azul de abajo:



Esta es una imagen de una parábola módulo 11. De hecho, dado que solo nos importan las entradas de $\mathbb{Z}_{11}$ para f, podrías decir con razón que **solo los 11 puntos resaltados** (no la curva azul) son una imagen de una parábola módulo 11.

17.5. Compartición de secretos de Shamir

Parte del desafío al diseñar un esquema de compartición de secretos es asegurarse de que *cualquier* conjunto autorizado de usuarios pueda reconstruir el secreto. Acabamos de ver que *cualesquiera* $d+1$ puntos en un polinomio de grado d son suficientes para reconstruir unívocamente el polinomio. Así que un enfoque natural para la compartición de secretos es dejar que la parte de cada usuario sea un punto en un polinomio.

Eso es exactamente lo que hace la compartición de secretos de Shamir. Para compartir un secreto $m \in \mathbb{Z}_p$ con un umbral t, primero se elige un polinomio f de grado $(t-1)$ que satisfaga $f(0) \equiv_p m$, con todos los demás coeficientes elegidos uniformemente en $\mathbb{Z}_p$. El i-ésimo usuario recibe el punto $(i, f(i) \mod p)$ en el polinomio. El teorema de interpolación dice que *cualesquiera* t de las partes pueden determinar unívocamente el polinomio f, y por lo tanto recuperar el secreto $f(0)$.

Shamir SSS:

	Share(m):	Reconstruct($\{s_i \mid i \in U\}$):
$\mathcal{M} = \mathbb{Z}_p$	$f_1, \ldots, f_{t-1} \leftarrow \mathbb{Z}_p$	$f(x) :=$ el (único) polinomio mód p
p : primo	$f(x) := m + \sum_{j=1}^{t-1} f_j x^j$	interpolador de grado $t-1$
$n < p$	for $i = 1$ to n:	que pasa por los puntos
$t \leq n$	$s_i := (i, f(i) \mod p)$	$\{s_i \mid i \in U\}$
	$s = (s_1, \ldots, s_n)$	return $f(0)$

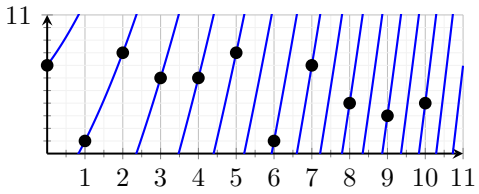
La corrección se deduce del teorema de interpolación.

Ejemplo 17.4. Aquí hay un ejemplo de compartición de secretos 3 de 5 sobre $\mathbb{Z}_{11}$ ($p = 11$). Supongamos que el secreto que se va a compartir es $m = 7 \in \mathbb{Z}_{11}$. El algoritmo Share elige un polinomio aleatorio de grado 2 con coeficiente constante 7.

Digamos que los dos coeficientes restantes se eligen como $f_2 = 1$ y $f_1 = 4$, lo que resulta en el siguiente polinomio:

$$f(x) = 1\,x^2 + 4\,x + 7$$

Este es el mismo polinomio ilustrado en el ejemplo anterior:



Para cada usuario $i \in \{1, ..., 5\}$, distribuimos la parte $(i, f(i) \mod 11)$. Estas partes corresponden a los puntos resaltados en la imagen módulo 11 de arriba.

usuario (i)	$f(i)$	parte (i, $f(i)$ mód 11)
1	$f(1) = 12$	(1, 1)
2	$f(2) = 19$	(2, 8)
3	$f(3) = 28$	(3, 6)
4	$f(4) = 39$	(4, 6)
5	$f(5) = 52$	(5, 8)

Recuerda que este ejemplo ilustra solo una posible ejecución de Share. Debido a que Share es un algoritmo aleatorizado, hay muchas comparticiones válidas del mismo secreto (inducidas por diferentes elecciones de los coeficientes resaltados en f).

17.5.1. Seguridad

Para mostrar la seguridad de la compartición de secretos de Shamir, primero mostramos un lema conveniente sobre la distribución de las partes en un conjunto no autorizado:

Lema 17.5.1. *Sea p un primo y definamos las siguientes dos librerías:*

$\mathcal{L}_{shamir\text{-}real}$

POLY$(m, t, U \subseteq \{1, \ldots, p\})$*:*
if $|U| \geq t$*: return err*
$f_1, ..., f_{t-1} \leftarrow \mathbb{Z}_p$
$f(x) := m + \sum_{j=1}^{t-1} f_j x^j$
for $i \in U$*:*
  $s_i := (i, f(i) \mod p)$
return $\{s_i \mid i \in U\}$

$\mathcal{L}_{shamir\text{-}rand}$

POLY$(m, t, U \subseteq \{1, \ldots, p\})$*:*
if $|U| \geq t$*: return err*
for $i \in U$*:*
  $y_i \leftarrow \mathbb{Z}_p$
  $s_i := (i, y_i)$
return $\{s_i \mid i \in U\}$

$\mathcal{L}_{shamir\text{-}real}$ elige un polinomio aleatorio de grado $(t-1)$ que pasa por el punto $(0, m)$, luego lo evalúa en las coordenadas x dadas (especificadas por U). $\mathcal{L}_{shamir\text{-}rand}$ simplemente da puntos elegidos uniformemente, no relacionados con ningún polinomio.

La afirmación es que estas librerías son intercambiables: $\mathcal{L}_{shamir\text{-}real} \equiv \mathcal{L}_{shamir\text{-}rand}$.

Demostración. Fijemos un mensaje $m \in \mathbb{Z}_p$, fijemos un conjunto U de usuarios con $|U| < t$, y para cada $i \in U$ fijemos un valor $y_i \in \mathbb{Z}_p$. Deseamos considerar la probabilidad de que una llamada a POLY(m, t, U) produzca $\{(i, y_i) \mid i \in U\}$, en cada una de las dos librerías.[2]

En la librerías $\mathcal{L}_{\text{shamir-real}}$, la subrutina elige un polinomio aleatorio de grado $(t-1)$ f tal que $f(0) \equiv_p m$. Del Corolario 17.4.3, sabemos que hay p^{t-1} tales polinomios.

Para que POLY produzca puntos consistentes con nuestros y_i elegidos, la librería debe haber elegido uno de los polinomios que pasa por $(0, m)$ y todos los puntos $\{(i, y_i) \mid i \in U\}$. La librería debe haber elegido uno de los polinomios que pasa a través de una elección específica de $|U| + 1$ puntos, y el Corolario 17.4.3 nos dice que hay $p^{t-(|U|+1)}$ tales polinomios.

La única forma de que POLY dé nuestra salida deseada es que elija uno de los $p^{t-(|U|+1)}$ polinomios "buenos", de las p^{t-1} posibilidades. Esto ocurre con una probabilidad exactamente igual a

$$\frac{p^{t-|U|-1}}{p^{t-1}} = p^{-|U|}$$

Ahora, en la librería $\mathcal{L}_{\text{shamir-rand}}$, POLY elige sus $|U|$ valores de salida uniformemente en $\mathbb{Z}_p$. Hay $p^{|U|}$ formas de elegirlos. Pero solo una de esas formas hace que POLY(m, t, U) produzca nuestra elección específica de $\{(i, y_i) \mid i \in U\}$. Por lo tanto, la probabilidad de recibir esta salida es $p^{-|U|}$.

Para todas las entradas posibles a POLY, ambas librerías asignan la misma probabilidad a cada salida posible. Por lo tanto, las librerías son intercambiables. ■

Teorema 17.5.2. *El esquema de compartición de secretos de Shamir es seguro de acuerdo con la Definición 17.2.1.*

Demostración. Sea $\mathcal{S}$ el esquema de compartición de secretos de Shamir. Probaremos que $\mathcal{L}_{\text{tsss-0}} \equiv \mathcal{L}_{\text{tsss-1}}$.

[2]Esto es similar a cómo, en la Afirmación 5.4.3, fijamos un m y c particulares y calculamos la probabilidad de que EAVESDROP$(m) = c$.

$\mathcal{L}^{\mathcal{S}}_{\text{tsss-0}}$

SHARE(m_0, m_1, U):
if $|U| \geq t$: return err
$f_1, \ldots, f_{t-1} \leftarrow \mathbb{Z}_p$
$f(x) := m_0 + \sum_{j=1}^{t-1} f_j x^j$
for $i \in U$:
  $s_i := (i, f(i) \mod p)$
return $\{s_i \mid i \in U\}$

Nuestro punto de partida es $\mathcal{L}^{\mathcal{S}}_{\text{tsss-0}}$, que se muestra aquí con los detalles de la compartición de secretos de Shamir rellenos.

SHARE(m_0, m_1, U):
return POLY(m_0, t, U)

$\diamond$

$\mathcal{L}_{\text{shamir-real}}$

POLY$(m, t, U \subseteq \{1, \ldots, p\})$:
if $|U| \geq t$: return err
$f_1, \ldots, f_{t-1} \leftarrow \mathbb{Z}_p$
$f(x) := m + \sum_{j=1}^{t-1} f_j x^j$
for $i \in U$:
  $s_i := (i, f(i) \mod p)$
return $\{s_i \mid i \in U\}$

Casi todo el cuerpo de la subrutina SHARE ha sido factorizado en términos de la librería $\mathcal{L}_{\text{shamir-real}}$ definida anteriormente. Lo único que queda es la "elección" de si compartir m_0 o m_1. Reestructurar el código de esta manera no tiene ningún efecto en el comportamiento de la librería.

SHARE(m_0, m_1, U):
return POLY(m_0, t, U)

$\diamond$

$\mathcal{L}_{\text{shamir-rand}}$

POLY$(m, t, U \subseteq \{1, \ldots, p\})$:
if $|U| \geq t$: return err
for $i \in U$:
  $y_i \leftarrow \mathbb{Z}_p$
  $s_i := (i, y_i)$
return $\{s_i \mid i \in U\}$

Mediante el Lema 17.5.1, podemos reemplazar $\mathcal{L}_{\text{shamir-real}}$ con $\mathcal{L}_{\text{shamir-rand}}$, sin que esto tenga ningún efecto en el comportamiento de la librería.

SHARE(m_0, m_1, U):
return POLY(m_1, t, U)

$\diamond$

$\mathcal{L}_{\text{shamir-rand}}$

POLY$(m, t, U \subseteq \{1, \ldots, p\})$:
if $|U| \geq t$: return err
for $i \in U$:
  $y_i \leftarrow \mathbb{Z}_p$
  $s_i := (i, y_i)$
return $\{s_i \mid i \in U\}$

El argumento a POLY ha sido cambiado de m_0 a m_1. Esto no tiene ningún efecto en el comportamiento de la librería, ya que POLY está ignorando su argumento.

$\underline{\text{SHARE}}(m_0, m_1, U)$:
return $\text{POLY}(m_1, t, U)$

$\diamond$

$\mathcal{L}_{\text{shamir-real}}$

$\underline{\text{POLY}}(m, t, U \subseteq \{1, \ldots, p\})$:
if $|U| \geq t$: return err
$f_1, ..., f_{t-1} \leftarrow \mathbb{Z}_p$
$f(x) := m + \sum_{j=1}^{t-1} f_j x^j$
for $i \in U$:
  $s_i := (i, f(i) \mod p)$
return $\{s_i \mid i \in U\}$

Aplicando los mismos pasos a la inversa, podemos reemplazar $\mathcal{L}_{\text{shamir-rand}}$ con $\mathcal{L}_{\text{shamir-real}}$, sin que esto tenga ningún efecto en el comportamiento de la librería.

$\mathcal{L}^{\mathcal{S}}_{\text{tsss-1}}$

$\underline{\text{SHARE}}(m_0, m_1, U)$:
if $|U| \geq t$: return err
$f_1, \ldots, f_{t-1} \leftarrow \mathbb{Z}_p$
$f(x) := m_1 + \sum_{j=1}^{t-1} f_j x^j$
for $i \in U$:
  $s_i := (i, f(i) \mod p)$
return $\{s_i \mid i \in U\}$

Se ha inlinado una subrutina, lo que no tiene ningún efecto en el comportamiento de la librería. La librería resultante es $\mathcal{L}^{\mathcal{S}}_{tsss-1}$.

Mostramos que $\mathcal{L}^{\mathcal{S}}_{\text{tsss-0}} \equiv \mathcal{L}_{\text{hyb-1}} \equiv \cdots \equiv \mathcal{L}_{\text{hyb-4}} \equiv \mathcal{L}^{\mathcal{S}}_{\text{tsss-1}}$, por lo que el esquema de compartición de secretos de Shamir es seguro.

■

Capítulo 18

Esquemas de compromiso

18.1. Introducción

La criptografía es, en muchos sentidos, un juego de confianza. Queremos que las partes involucradas en un protocolo puedan asegurarse mutuamente de que no están haciendo trampas, sin necesidad de confiar ciegamente. Uno de los problemas más básicos en este contexto es el de comprometerse con un valor sin revelarlo de inmediato, pero garantizando que no podrá cambiarse después. Aquí es donde entran en juego los **esquemas de compromiso**.

Un esquema de compromiso puede ser visto como un sobre digital: sellar un mensaje m en un sobre y dar el sobre a otra parte oculta m (hasta que se abra el sobre) a pesar de que el valor de m es fijo (ya que el contenido del sobre no se puede cambiar).

Supongamos que Alice desea jugar "piedra-papel-tijeras" por teléfono con Bob. La idea de este juego es que Alice y Bob eligen simultáneamente uno del conjunto {piedra, papel, tijeras}. Luego, el resultado del juego está determinado por las reglas:

- El papel envuelve la piedra. Por lo tanto, si Alice elige papel y Bob elige piedra, entonces Alice gana.
- La piedra aplasta las tijeras. Por lo tanto, si Alice elige piedra y Bob elige tijeras, entonces Alice gana.
- Las tijeras cortan el papel. Por lo tanto, si Alice elige tijeras y Bob elige papel, entonces Alice gana.

Si tanto Alice como Bob eligen el mismo elemento, entonces el juego se declara empate. Cuando se realiza por teléfono, tenemos el problema de que quien vaya primero va a perder el juego.

Una forma de evitar esto es que la parte que va primero se "comprometa" con su elección, de tal manera que la otra parte no pueda determinar a qué se comprometió. Luego, las dos partes pueden revelar sus elecciones, con la idea de que la otra parte pueda verificar que la parte que revela no ha alterado su elección

entre la etapa de compromiso y la de revelación. Tal sistema se llama esquema de compromiso. Una forma fácil de hacer esto es usar una función hash criptográfica de la siguiente manera:

$$A \longrightarrow B : h_A = H(R_A||\text{papel})$$
$$B \longrightarrow A : \text{tijeras}$$
$$A \longrightarrow B : R_A, \text{papel}$$

Al final del protocolo, Bob necesita verificar que el h_A enviado por Alice es igual a $H(R_A||\text{papel})$. Si los valores coinciden, él sabe que Alice no ha hecho trampa. El resultado de este protocolo es que Alice pierde el juego ya que las tijeras cortan el papel.

Veamos lo anterior desde la perspectiva de Alice. Ella primero se compromete con el valor *papel* enviando a Bob el valor hash h_A. Esto significa que Bob no podrá determinar que Alice se ha comprometido con el valor *papel*, ya que Bob no conoce el valor aleatorio R_A utilizado y Bob es incapaz de invertir la función hash. El hecho de que Bob no pueda determinar con qué valor se comprometió se llama la propiedad de *ocultación o confidencialidad* de un esquema de compromiso. El valor aleatorio R_A se utiliza para evitar ataques de precomputación. Sin él, un atacante podría precalcular los valores hash para todas las posibles elecciones y luego simplemente revelar la elección que le da la victoria.

Tan pronto como Bob envía el valor *tijeras* a Alice, ella sabe que ha perdido, pero es incapaz de hacer trampa, ya que para hacer trampa necesitaría idear un valor diferente de R_A, digamos R'_A, que satisfaga

$$H(R_A||\text{papel}) = H(R'_A||\text{piedra}).$$

Pero esto significaría que Alice podría encontrar colisiones en la función hash, lo cual para una función hash elegida adecuadamente se cree que es imposible. En realidad, requerimos que la función hash sea resistente a la segunda preimagen en este caso. Esto significa que, dado un valor hash y un valor de entrada (en este caso, R_A y "papel"), es computacionalmente inviable encontrar otro valor de entrada (R'_A y "piedra") que produzca el mismo valor hash. Esta propiedad del esquema de compromiso, que Alice no puede cambiar de opinión después del procedimiento de compromiso, se llama *vinculación*. El uso de una función hash criptográfica asegura que es computacionalmente inviable para un jugador cambiar su elección después de comprometerse con ella.

Este ejemplo muestra la esencia de los esquemas de compromiso: permiten fijar un valor sin revelarlo de inmediato y garantizan que no pueda cambiarse después. Son herramientas fundamentales en muchas áreas de la criptografía, como protocolos de conocimiento cero, votaciones electrónicas y protocolos de firma digital.

18.2. Definición Formal y Seguridad

Los esquemas de compromiso se diseñan siguiendo un enfoque en dos fases:

1. **Fase de compromiso:** Todas las entidades interesadas envían su propuesta de forma oculta, asegurando que ningún participante pueda modificar su elección después de enviarla.
2. **Fase de apertura:** Cada participante revela su propuesta junto con el parámetro necesario para verificar que esta coincide con la enviada en la primera fase. Esto garantiza que el proceso sea justo y

transparente, permitiendo determinar sin ambigüedades quién tiene la propuesta más ventajosa o, en el caso del ejemplo anterior, quién ha ganado el juego de "piedra-papel-tijera".

Es importante destacar que, a diferencia de otros esquemas criptográficos, en este contexto la funcionalidad de "cifrado y descifrado" no es esencial. En lugar de una clave secreta utilizada para recuperar información oculta, el compromiso se protege mediante la aleatoriedad empleada en la fase inicial, la cual se hace pública en la fase de apertura. De este modo, la seguridad del esquema se basa en la imposibilidad de modificar una propuesta después de enviarla, sin necesidad de recurrir a técnicas de cifrado tradicionales.

Así, un esquema de compromiso se define de la siguiente manera.

Definición 18.2.1 (Esquema de Compromiso)**.** *Sea $\mathcal{K}$ un espacio de claves de compromiso, $\mathcal{M}$ un espacio de mensajes, $\mathcal{C}$ un espacio de compromisos y $\mathcal{R}$ un espacio de valores de apertura. Un esquema de compromiso consta de tres algoritmos probabilísticos:*

- ***Inicialización:*** $\mathsf{Setup} : \mathbb{N} \to \mathcal{K}$*, que toma como entrada un parámetro de seguridad y devuelve una clave de compromiso $ck \in \mathcal{K}$, que se hace pública.*
- ***Compromiso:*** $\mathsf{Com} : \mathcal{M} \times \mathcal{K} \to \mathcal{C} \times \mathcal{R}$*, que toma un mensaje $m \in \mathcal{M}$ y una clave de compromiso $ck \in \mathcal{K}$ y da como salida dos valores: el compromiso $c = \mathsf{Com}(ck, m)$ y un valor de apertura $d \in \mathcal{R}$.*
- ***Apertura:*** $\mathsf{Open} : \mathcal{K} \times \mathcal{R} \times \mathcal{C} \to \mathcal{M}$*, que, recibiendo una clave de compromiso ck, un valor de apertura d y un compromiso c, da como salida un mensaje m si el compromiso es válido o un mensaje de error en caso contrario.*

La seguridad de un esquema de compromiso se define a través de dos propiedades fundamentales, a saber, ocultación y vinculación. Una propiedad protege los intereses del remitente (ocultamiento) y otra propiedad protege los intereses del receptor (vinculación).

Definición 18.2.2 (Ocultación)**.** *Sea Σ un esquema de compromiso. Decimos que Σ tiene la propiedad de* ***ocultación*** *si $\mathcal{L}^{\Sigma}_{hiding\text{-}0} \approx \mathcal{L}^{\Sigma}_{hiding\text{-}1}$, donde:*

$\mathcal{L}^{\Sigma}_{hiding\text{-}0}$

$ck \leftarrow \Sigma.\mathsf{Setup}(n)$

GETCK():
return ck

$HIDING(m_0, m_1 \in \Sigma.\mathcal{M})$*:*
if $|m_0| \neq |m_1|$*: return err*
$c \leftarrow \Sigma.\mathsf{Com}(ck, m_0)$
return c

$\mathcal{L}^{\Sigma}_{hiding\text{-}1}$

$ck \leftarrow \Sigma.\mathsf{Setup}(n)$

GETCK():
return ck

$HIDING(m_0, m_1 \in \Sigma.\mathcal{M})$*:*
if $|m_0| \neq |m_1|$*: return err*
$c \leftarrow \Sigma.\mathsf{Com}(ck, m_1)$
return c

Observa que esta definición es prácticamente idéntica a la definición de seguridad CPA para criptografía de clave pública (Definición 16.1.1). Por lo tanto, **ningún esquema determinista tiene la propiedad de ocultación**.

Definición 18.2.3 (Vinculación). *Sea Σ un esquema de compromiso. Decimos que Σ tiene la propiedad de* ***vinculación*** *si $\mathcal{L}^{\Sigma}_{binding\text{-}real} \approx \mathcal{L}^{\Sigma}_{binding\text{-}fake}$, donde:*

$\mathcal{L}^{\Sigma}_{binding\text{-}real}$

$ck \leftarrow \Sigma.\mathsf{Setup}(n)$

GETCK():
return ck

BINDING$(c \in \Sigma.\mathcal{C}, d, d' \in \Sigma.\mathcal{R})$*:*
$m \leftarrow \Sigma.\mathsf{Open}(ck, c, d)$
$m' \leftarrow \Sigma.\mathsf{Open}(ck, c, d')$
if $m \neq m'$ *and* $\Sigma.\mathsf{Com}(ck, m) = c = \Sigma.\mathsf{Com}(ck, m')$
  return true
return false

$\approx$

$\mathcal{L}^{\Sigma}_{binding\text{-}fake}$

$ck \leftarrow \Sigma.\mathsf{Setup}(n)$

GETCK():
return ck

BINDING$(c \in \Sigma.\mathcal{C}, d, d' \in \Sigma.\mathcal{R})$*:*
return false

Esta definición es prácticamente igual a la definición de resistencia a colisiones de una función hash (Definición 13.1.1).

En el juego de piedra-papel-tijera mencionado en la introducción, señalamos que bastaba con que la función hash fuera resistente a segundas preimágenes para impedir que Alice cambiara de opinión, es decir, para garantizar que no pudiera desvincularse de su decisión inicial. Sin embargo, nuestra definición de vinculación requiere que la función hash sea resistente a colisiones, lo que, a su vez, implica resistencia a segundas preimágenes.

18.3. Esquemas de compromiso basado en una función hash

La construcción que presentaremos a continuación se basa en modelo idealizado de las funciones hash. Esta idealización consiste en que las salidas de una función hash $H : \{0,1\}^* \to \{0,1\}^n$ son indistinguibles de elementos escogidos uniformemente al azar. Formalizamos esta noción en la siguiente definición.

Definición 18.3.1 (Hash con salidas indistiguibles de uniforme). *Una función hash $H : \{0,1\}^* \to \{0,1\}^n$ con* ***salidas indisinguibles de uniforme*** *satisface $\mathcal{L}^{H}_{hash\text{-}real} \equiv \mathcal{L}^{H}_{hash\text{-}rand}$, donde:*

$\mathcal{L}^{H}_{hash\text{-}real}$

$s \leftarrow \{0,1\}^{\lambda}$

GETSALT():
return s

ORACLE$(x \in \{0,1\}^*)$*:*
return $H(s, x)$

$\mathcal{L}^{H}_{hash\text{-}rand}$

$s \leftarrow \{0,1\}^{\lambda}$

GETSALT():
return s

ORACLE$(x \in \{0,1\}^*)$*:*
$y \leftarrow \{0,1\}^n$
return y

Una función con salidas indistinguibles de uniforme es una idealización que no puede implementarse en la práctica. Sin embargo, su utilidad radica en que permite desarrollar pruebas de concepto y analizar posibles vulnerabilidades en el diseño de esquemas criptográficos.

Sea $H : \{0,1\}^* \to \{0,1\}^n$ una función hash con salidas indistinguibles de uniforme. Podemos construir el siguiente esquema de compromiso.

Compromiso basado en hash:

$\mathcal{K} = \{0,1\}^\lambda$	$\underline{\mathsf{Setup}(\lambda)}$:	$\underline{\mathsf{Com}(ck, m)}$:	$\underline{\mathsf{Open}(ck, m, c)}$:
$\mathcal{M} = \{0,1\}^*$	$ck \leftarrow \{0,1\}^\lambda$	return $H(ck\|\|m)$	if $c \neq H(ck\|\|m)$: return error
$\mathcal{C} = \{0,1\}^n$			return m

El valor de apertura en este esquema de compromiso se ha definido de manera implícita. Para entenderlo mejor, observemos las entradas del algoritmo Open, que corresponden a la terna (ck, m, c). Aquí, tanto la clave de compromiso ck como el compromiso c son valores públicos obtenidos a partir de la ejecución de los algoritmos Setup y Com.

Para ejecutar Open, es necesario revelar el mensaje m. Es importante recordar que el objetivo de un esquema de compromiso no es ocultar el mensaje indefinidamente, sino posponer su revelación hasta el momento adecuado. Por ejemplo, en el juego de piedra-papel-tijera, la elección de cada jugador se hace pública en cada ronda, pero no antes de la fase de apertura del compromiso.

En el caso del esquema de compromiso basado en funciones hash, el mensaje m también se revela en la fase de apertura. Esto significa que el propio mensaje actúa como el valor de apertura, permitiendo verificar que el compromiso original fue generado correctamente.

Teorema 18.3.2 (Seguridad del compromiso basado en hash)**.** *Si H es una función hash resistente a colisiones y sus salidas son indistinguibles de uniforme, entonces el esquema satisface las propiedades de ocultación y vinculación.*

Demostración. Primero veremos que el esquema de compromiso tiene la propiedad de ocultación demostrando que $\mathcal{L}^{\Sigma}_{\text{hiding-0}} \approx \mathcal{L}^{\Sigma}_{\text{hiding-1}}$.

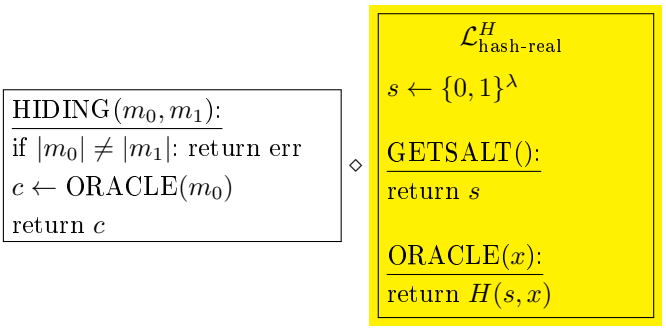
$\mathcal{L}^{\Sigma}_{\text{hiding-0}}$

$ck \leftarrow \{0,1\}^\lambda$

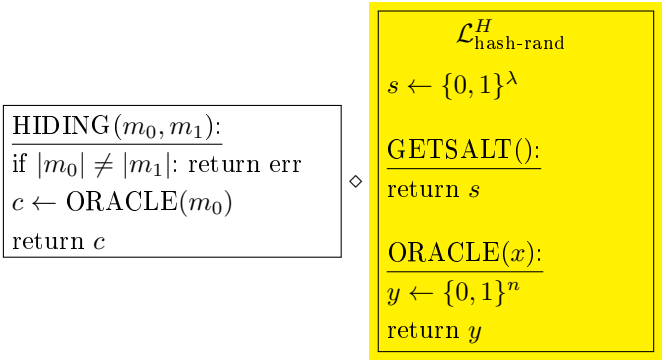
$\underline{\text{GETCK}()}$:
return ck

$\underline{\text{HIDING}(m_0, m_1)}$:
if $|m_0| \neq |m_1|$: return err
$c \leftarrow H(ck\|m_0)$
return c

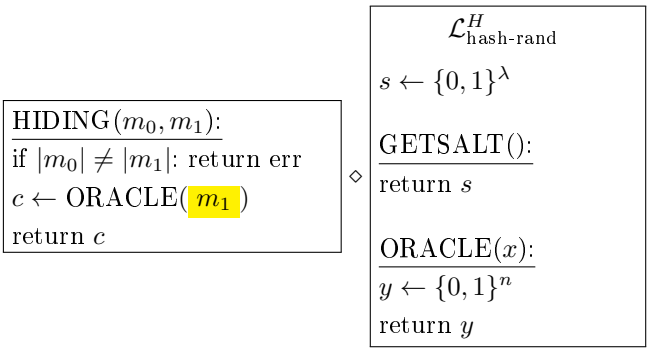
Nuestro punto de partida es $\mathcal{L}^{\Sigma}_{\text{hiding-0}}$, que se muestra aquí con los detalles del esquema de compromiso.



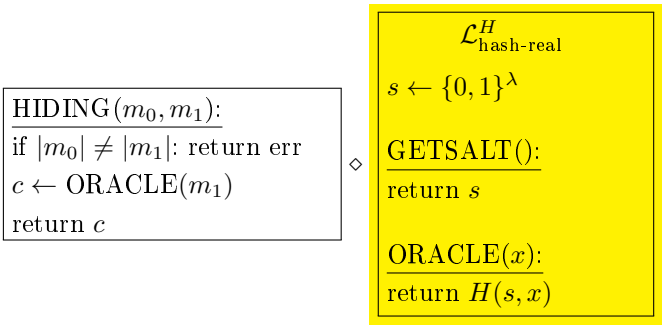
Casi todo el cuerpo de la librería $\mathcal{L}^{\Sigma}_{\text{hiding-0}}$ ha sido factorizado en términos de la librería $\mathcal{L}^H_{\text{hash-real}}$ definida anteriormente. Lo único que queda es la "elección" de si compartir m_0 o m_1. Reestructurar el código de esta manera no tiene ningún efecto en el comportamiento de la librería.



Por la indistinguibilidad de las salidas de la función hash, podemos reemplazar $\mathcal{L}^H_{\text{hash-real}}$ con $\mathcal{L}^H_{\text{hash-rand}}$, sin que esto tenga ningún efecto en el comportamiento de la librería.



El argumento a HIDING ha sido cambiado de m_0 a m_1. Esto no tiene ningún efecto en el comportamiento de la librería, ya que HIDING está ignorando su argumento.



Aplicando los mismos pasos a la inversa, podemos reemplazar $\mathcal{L}^H_{\text{hash-rand}}$ con $\mathcal{L}^H_{\text{hash-real}}$, sin que esto tenga ningún efecto en el comportamiento de la librería.

$\mathcal{L}^{\Sigma}_{\text{hiding-1}}$

$ck \leftarrow \{0,1\}^{\lambda}$

GETCK():
return ck

HIDING(m_0, m_1):
if $|m_0| \neq |m_1|$: return err
$c \leftarrow H(ck \| m_1)$
return c

Se ha inlinado una subrutina, lo que no tiene ningún efecto en el comportamiento de la librería. La librería resultante es $\mathcal{L}^{\Sigma}_{\text{hiding-1}}$.

Mostramos que $\mathcal{L}^{\Sigma}_{\text{hiding-0}} \approx \mathcal{L}_{\text{hyb-1}} \equiv \cdots \equiv \mathcal{L}_{\text{hyb-4}} \approx \mathcal{L}^{\Sigma}_{\text{hiding-1}}$, por lo que el esquema de compromiso tiene la propiedad de ocultación.

Para ver que este esquema de compromiso tiene la propiedad de vinculación, demostraremos que $\mathcal{L}^{\Sigma}_{\text{binding-real}} \approx \mathcal{L}^{\Sigma}_{\text{binding-fake}}$.

$\mathcal{L}^{\Sigma}_{\text{binding-real}}$

$ck \leftarrow \{0,1\}^{\lambda}$

GETCK():
return ck

BINDING(c, m, m'):
$m \leftarrow \mathsf{Open}(ck, m, c)$,
$m' \leftarrow \mathsf{Open}(ck, m', c)$
if $m \neq m'$ and $H(ck \| m) = c = H(ck \| m')$
    return true
return false

Nuestro punto de partida es $\mathcal{L}^{\Sigma}_{\text{binding-real}}$, que se muestra aquí con los detalles del esquema de compromiso.

$ck \leftarrow$ GETSALT()

BINDING(c, m, m'):
$m \leftarrow \mathsf{Open}(ck, m, c)$,
$m' \leftarrow \mathsf{Open}(ck, m', c)$
TEST(m, m')

$\diamond$

$\mathcal{L}^{H}_{\text{cr-real}}$

$s \leftarrow \{0,1\}^{\lambda}$

GETSALT():
return s

TEST(x, x'):
if $x \neq x'$ and $H(s \| x) = H(s \| x')$
    return true
return false

Casi todo el cuerpo de la librería $\mathcal{L}^{\Sigma}_{\text{binding-real}}$ ha sido factorizado en términos de la librería $\mathcal{L}^{H}_{\text{cr-real}}$.

$ck \leftarrow \text{GETSALT}()$

$\underline{\text{BINDING}(c, m, m')}$:
$m \leftarrow \mathsf{Open}(ck, m, c)$,
$m' \leftarrow \mathsf{Open}(ck, m', c)$
$\text{TEST}(m, m')$

$\diamond$

$\mathcal{L}^{H}_{\text{cr-fake}}$

$s \leftarrow \{0,1\}^{\lambda}$

$\underline{\text{GETSALT}()}$:
return s

$\underline{\text{TEST}(x, x')}$:
return false

La función hash H utilizada es resistente a colisiones, así que podemos reemplazar $\mathcal{L}^{H}_{\text{cr-real}}$ con $\mathcal{L}^{H}_{\text{cr-fake}}$, sin que esto tenga ningún efecto en el comportamiento de la librería.

$ck \leftarrow \{0,1\}^{\lambda}$

$\underline{\text{GETCK}()}$:
return ck

$\underline{\text{BINDING}(c, m, m')}$:
$m \leftarrow \mathsf{Open}(ck, m, c)$,
$m' \leftarrow \mathsf{Open}(ck, m', c)$
return false

Se ha inlinado una subrutina, lo que no tiene ningún efecto en el comportamiento de la librería.

$\mathcal{L}^{\Sigma}_{\text{binding-fake}}$

$ck \leftarrow \{0,1\}^{\lambda}$

$\underline{\text{GETCK}()}$:
return ck

$\underline{\text{BINDING}(c, m, m')}$:
return false

La subrutina BINDING ignora las llamadas a Open, así que borrarlas no tiene ningún efecto en el comportamiento de la librería. La librería resultante es $\mathcal{L}^{\Sigma}_{\text{binding-real}}$.

Hemos mostrado que $\mathcal{L}^{\Sigma}_{\text{binding-real}} \equiv \mathcal{L}_{\text{hyb-1}} \approx \mathcal{L}_{\text{hyb-2}} \equiv \mathcal{L}_{\text{hyb-3}} \equiv \mathcal{L}^{\Sigma}_{\text{binding-fake}}$, por lo que el esquema de compromiso tiene la propiedad de vinculación.

■

Librerías

Cifrados uniformes / psuedouniformes de un solo uso:

$\mathcal{L}^{\Sigma}_{\text{otu}}$

$\underline{\text{CTXT}(m \in \Sigma.\mathcal{M}):}$
$k \leftarrow \Sigma.\text{Gen}$
$c := \Sigma.\text{Enc}(k, m)$
return c

$\mathcal{L}^{\Sigma}_{\text{otu-\$}}$

$\underline{\text{CTXT}(m \in \Sigma.\mathcal{M}):}$
$c \leftarrow \Sigma.\mathcal{C}$
return c

Intercambiabilidad / Indistiguibilidad de un solo uso:

$\mathcal{L}^{\Sigma}_{\text{otint-0}}$

$\underline{\text{EAVESDROP}(m_0, m_1 \in \Sigma.\mathcal{M}):}$
$k \leftarrow \Sigma.\text{Gen}$
$c := \Sigma.\text{Enc}(k, m_0)$
return c

$\mathcal{L}^{\Sigma}_{\text{otint-1}}$

$\underline{\text{EAVESDROP}(m_0, m_1 \in \Sigma.\mathcal{M}):}$
$k \leftarrow \Sigma.\text{Gen}$
$c := \Sigma.\text{Enc}(k, m_1)$
return c

PRG seguro:

$\mathcal{L}^{G}_{\text{prg}}$

$\underline{\text{QUERY}():}$
$s \leftarrow \{0,1\}^{\lambda}$
return $G(s)$

$\mathcal{L}^{G}_{\text{prg-\$}}$

$\underline{\text{QUERY}():}$
$r \leftarrow \{0,1\}^{\lambda+\ell}$
return r

PRF segura:

$\mathcal{L}^{F}_{\text{prf}}$

$k \leftarrow \{0,1\}^{\lambda}$

$\underline{\text{LOOKUP}(x \in \{0,1\}^{in}):}$
return $F(k, x)$

$\mathcal{L}^{F}_{\text{prf-\$}}$

$T :=$ array asoc. vacío

$\underline{\text{LOOKUP}(x \in \{0,1\}^{in}):}$
if $T[x]$ no está definido:
  $T[x] \leftarrow \{0,1\}^{out}$
return $T[x]$

Seguridad CPA:

$\mathcal{L}^{\Sigma}_{\text{cpa-0}}$

$k \leftarrow \Sigma.\text{Gen}$

EAVESDROP$(m_0, m_1 \in \Sigma.\mathcal{M})$:
$c := \Sigma.\text{Enc}(k, m_0)$
return c

$\mathcal{L}^{\Sigma}_{\text{cpa-1}}$

$k \leftarrow \Sigma.\text{Gen}$

EAVESDROP$(m_0, m_1 \in \Sigma.\mathcal{M})$:
$c := \Sigma.\text{Enc}(k, m_1)$
return c

Seguridad pCPA:

$\mathcal{L}^{\Sigma}_{\text{pCPA}}$

$k \leftarrow \Sigma.\text{Gen}$

CTXT$(m \in \Sigma.\mathcal{M})$:
$c := \Sigma.\text{Enc}(k, m)$
return c

$\mathcal{L}^{\Sigma}_{\text{pCPA-\$}}$

CTXT$(m \in \Sigma.\mathcal{M})$:
$c \leftarrow \Sigma.\mathcal{C}$
return c

Seguridad CCA:

$\mathcal{L}^{\Sigma}_{\text{cca-0}}$

$k \leftarrow \Sigma.\text{Gen}$
$\mathcal{S} := \emptyset$

EAVESDROP$(m_0, m_1 \in \Sigma.\mathcal{M})$:
If $|m_0| \neq |m_1|$ return error
$c := \Sigma.\text{Enc}(k, m_0)$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

DECRYPT$(c \in \Sigma.\mathcal{C})$:
If $c \in \mathcal{S}$ return error
return $\Sigma.\text{Dec}(k, c)$

$\mathcal{L}^{\Sigma}_{\text{cca-1}}$

$k \leftarrow \Sigma.\text{Gen}$
$\mathcal{S} := \emptyset$

EAVESDROP$(m_0, m_1 \in \Sigma.\mathcal{M})$:
If $|m_0| \neq |m_1|$ return error
$c := \Sigma.\text{Enc}(k, m_1)$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

DECRYPT$(c \in \Sigma.\mathcal{C})$:
If $c \in \mathcal{S}$ return error
return $\Sigma.\text{Dec}(k, c)$

Seguridad pCCA:

$\mathcal{L}^{\Sigma}_{\text{pCCA}}$

$k \leftarrow \Sigma.\text{Gen}$
$\mathcal{S} := \emptyset$

CTXT$(m \in \Sigma.\mathcal{M})$:
$c := \Sigma.\text{Enc}(k, m)$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

DECRYPT$(c \in \Sigma.\mathcal{C})$:
If $c \in \mathcal{S}$ return error
return $\Sigma.\text{Dec}(k, c)$

$\mathcal{L}^{\Sigma}_{\text{pCCA-\$}}$

$k \leftarrow \Sigma.\text{Gen}$
$\mathcal{S} := \emptyset$

CTXT$(m \in \Sigma.\mathcal{M})$:
$c := \Sigma.\mathcal{C}(|m|)$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

DECRYPT$(c \in \Sigma.\mathcal{C})$:
If $c \in \mathcal{S}$ return error
return $\Sigma.\text{Dec}(k, c)$

Seguridad MAC:

$\mathcal{L}^{\Sigma}_{\text{mac-real}}$

$k \leftarrow \Sigma.\text{Gen}$

GETTAG$(m \in \Sigma.\mathcal{M})$:
return $\Sigma.\text{MAC}(k, m)$

CHECKTAG$(m \in \Sigma.\mathcal{M}, t)$:
return $t \stackrel{?}{=} \Sigma.\text{MAC}(k, m)$

$\mathcal{L}^{\Sigma}_{\text{mac-fake}}$

$k \leftarrow \Sigma.\text{Gen}$
$\mathcal{T} := \emptyset$

GETTAG$(m \in \Sigma.\mathcal{M})$:
$t := \Sigma.\text{MAC}(k, m)$
$\mathcal{T} := \mathcal{T} \cup \{(m, t)\}$
return t

CHECKTAG$(m \in \Sigma.\mathcal{M}, t)$:
return $(m, t) \stackrel{?}{\in} \mathcal{T}$

Seguridad firmas digitales:

$\mathcal{L}^{\Sigma}_{\text{sig-real}}$

$(vk, sk) \leftarrow \Sigma.\text{Gen}$

GETVK():
return vk

GETSIG(m):
return $\Sigma.\texttt{Sign}(sk, m)$

VERSIG(m, σ):
return $\Sigma.\texttt{Ver}(vk, m, \sigma)$

$\mathcal{L}^{\Sigma}_{\text{sig-fake}}$

$(vk, sk) \leftarrow \Sigma.\text{Gen}$
$\mathcal{S} := \emptyset$

GETVK():
return vk

GETSIG(m):
$\sigma := \Sigma.\texttt{Sign}(sk, m)$
$\mathcal{S} := \mathcal{S} \cup \{(m, \sigma)\}$
return σ

VERSIG(m, σ):
return $(m, \sigma) \stackrel{?}{\in} \mathcal{S}$

Seguridad establecimiento de claves:

$\mathcal{L}^{\Sigma}_{\text{ka}}$

QUERY():
$(t, K) \leftarrow \text{EXECPROT}(\Sigma)$
return (t, K)

$\mathcal{L}^{\Sigma}_{\text{ka-\$}}$

QUERY():
$(t, K) \leftarrow \text{EXECPROT}(\Sigma)$
$K' \leftarrow \Sigma.\mathcal{K}$
return (t, K')

Diffie-Hellman Decisional:

$\mathcal{L}^{\mathbb{G}}_{\text{dh}}$

QUERY():
$a, b \leftarrow \mathbb{Z}_n$
return (g^a, g^b, g^{ab})

$\mathcal{L}^{\mathbb{G}}_{\text{dh-\$}}$

QUERY():
$a, b, c \leftarrow \mathbb{Z}_n$
return (g^a, g^b, g^c)

Seguridad CPA de clave pública:

$\mathcal{L}^{\Sigma}_{\text{pk-cpa-0}}$

$(pk, sk) \leftarrow \Sigma.\text{Gen}$

GETPK():
return pk

CHALLENGE($m_0, m_1 \in \Sigma.\mathcal{M}$):
return $\Sigma.\text{Enc}(pk, m_0)$

$\mathcal{L}^{\Sigma}_{\text{pk-cpa-1}}$

$(pk, sk) \leftarrow \Sigma.\text{Gen}$

GETPK():
return pk

CHALLENGE($m_0, m_1 \in \Sigma.\mathcal{M}$):
return $\Sigma.\text{Enc}(pk, m_1)$

Seguridad pCPA de clave pública:

$\mathcal{L}^{\Sigma}_{\text{pk-pCPA}}$

$(pk, sk) \leftarrow \Sigma.\text{Gen}$

GETPK():
return pk

CHALLENGE($m \in \Sigma.\mathcal{M}$):
return $\Sigma.\text{Enc}(pk, m)$

$\mathcal{L}^{\Sigma}_{\text{pk-pCPA-\$}}$

$(pk, sk) \leftarrow \Sigma.\text{Gen}$

GETPK():
return pk

CHALLENGE($m \in \Sigma.\mathcal{M}$):
$c \leftarrow \Sigma.\mathcal{C}$
return c

Seguridad CCA de clave pública:

$\mathcal{L}^{\Sigma}_{\text{pk-cca-0}}$

$(pk, sk) \leftarrow \Sigma.\text{Gen}$
$\mathcal{S} := \emptyset$

GETPK():
return pk

CHALLENGE($m_0, m_1 \in \Sigma.\mathcal{M}$):
If $|m_0| \neq |m_1|$ return error
$c := \Sigma.\text{Enc}(pk, m_0)$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

DECRYPT($c \in \Sigma.\mathcal{C}$):
If $c \in \mathcal{S}$ return error
return $\Sigma.\text{Dec}(sk, c)$

$\mathcal{L}^{\Sigma}_{\text{pk-cca-1}}$

$(pk, sk) \leftarrow \Sigma.\text{Gen}$
$\mathcal{S} := \emptyset$

GETPK():
return pk

CHALLENGE($m_0, m_1 \in \Sigma.\mathcal{M}$):
If $|m_0| \neq |m_1|$ return error
$c := \Sigma.\text{Enc}(pk, m_1)$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

DECRYPT($c \in \Sigma.\mathcal{C}$):
If $c \in \mathcal{S}$ return error
return $\Sigma.\text{Dec}(sk, c)$

Seguridad pCCA de clave pública:

$\mathcal{L}^{\Sigma}_{\text{pk-pCCA}}$

$(pk, sk) \leftarrow \Sigma.\text{Gen}$
$\mathcal{S} := \emptyset$

GETPK():
return pk

CHALLENGE($m \in \Sigma.\mathcal{M}$):
$c := \Sigma.\text{Enc}(pk, m)$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

DECRYPT($c \in \Sigma.\mathcal{C}$):
If $c \in \mathcal{S}$ return error
return $\Sigma.\text{Dec}(sk, c)$

$\mathcal{L}^{\Sigma}_{\text{pk-pCCA-\$}}$

$(pk, sk) \leftarrow \Sigma.\text{Gen}$
$\mathcal{S} := \emptyset$

GETPK():
return pk

CHALLENGE($m \in \Sigma.\mathcal{M}$):
$c := \Sigma.\mathcal{C}(|m|)$
$\mathcal{S} := \mathcal{S} \cup \{c\}$
return c

DECRYPT($c \in \Sigma.\mathcal{C}$):
If $c \in \mathcal{S}$ return error
return $\Sigma.\text{Dec}(sk, c)$

Maleabilidad CPA:

$\mathcal{L}^{\Sigma}_{\text{mCPA}}$

$(pk, sk) := \Sigma.\text{Gen}$
$B := \emptyset$

GETPK():
return pk

MCTXT($\ell; m_1, \ldots, m_\ell \in \Sigma.\mathcal{M}$):
for $i = 1$ to ℓ
  $B := B \cup \{m_i\}$
  $c_i := \Sigma.\text{Enc}(pk, m_i)$
return $c_1, \ldots, c_\ell$

VERIFY($c', \mathcal{R}$):
$m' := \Sigma.\text{Dec}(sk, c')$
If $m' \notin \Sigma.\mathcal{M}$
  return error
return $\mathcal{R}(B, m') \stackrel{?}{=} 1$

Seguridad de compartición de secretos:

$\mathcal{L}^{\Sigma}_{\text{tsss-0}}$

SHARE($m_0, m_1 \in \Sigma.\mathcal{M}, U$):
if $|U| \geq \Sigma.t$: return err
$s \leftarrow \Sigma.\text{Share}(m_0)$
return $\{s_i \mid i \in U\}$

$\mathcal{L}^{\Sigma}_{\text{tsss-1}}$

SHARE($m_0, m_1 \in \Sigma.\mathcal{M}, U$):
if $|U| \geq \Sigma.t$: return err
$s \leftarrow \Sigma.\text{Share}(m_1)$
return $\{s_i \mid i \in U\}$

Seguridad hiding:

$\mathcal{L}^{\Sigma}_{\text{hiding-0}}$

$ck \leftarrow \Sigma.\mathsf{Setup}(n)$

GETCK():
return ck

HIDING($m_0, m_1 \in \Sigma.\mathcal{M}$):
if $|m_0| \neq |m_1|$: return err
$c \leftarrow \Sigma.\mathsf{Com}(ck, m_0)$
return c

$\mathcal{L}^{\Sigma}_{\text{hiding-1}}$

$ck \leftarrow \Sigma.\mathsf{Setup}(n)$

GETCK():
return ck

HIDING($m_0, m_1 \in \Sigma.\mathcal{M}$):
if $|m_0| \neq |m_1|$: return err
$c \leftarrow \Sigma.\mathsf{Com}(ck, m_1)$
return c

Seguridad binding:

$\mathcal{L}^{\Sigma}_{\text{binding-real}}$

$ck \leftarrow \Sigma.\mathsf{Setup}(n)$

GETCK():
return ck

BINDING($c \in \Sigma.\mathcal{C}, d, d' \in \Sigma.\mathcal{R}$):
$m \leftarrow \Sigma.\mathsf{Open}(ck, c, d)$
$m' \leftarrow \Sigma.\mathsf{Open}(ck, c, d')$
if $m \neq m'$ and $\Sigma.\mathsf{Com}(ck, m) = c = \Sigma.\mathsf{Com}(ck, m')$
    return true
return false

$\mathcal{L}^{\Sigma}_{\text{binding-fake}}$

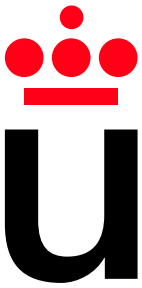
$ck \leftarrow \Sigma.\mathsf{Setup}(n)$

GETCK():
return ck

BINDING($c \in \Sigma.\mathcal{C}, d, d' \in \Sigma.\mathcal{R}$):
return false

PARTE III

Ejercicios resueltos (Categoría 3)



Universidad
Rey Juan Carlos

Ejercicios resueltos
CRIPTOGRAFÍA

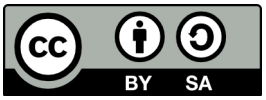
Grado en Ingeniería de la Ciberseguridad

Misael E. Marriaga

Material docente en abierto de la Universidad Rey Juan Carlos

Asignatura:	Criptografía
Titulación:	Grado en Ingeniería de la Ciberseguridad
Código:	2285007
Curso:	2025-2026

Autoría:
Misael E. Marriaga
misael.marriaga@urjc.es



©2026 Misael E. Marriaga
Algunos derechos reservados
Este documento se distribuye bajo una licencia Creative Commons, Atribución-CompartirIgual 4.0 Internacional (CC BY-SA 4.0), disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Criptografía – Grado en Ingeniería de la Ciberseguridad

Hoja de problemas 1: Intercambiabilidad de librerías y distinguidoras

Escuela de Ingeniería Informática

Objetivos.

1. Dominar el lenguaje de **librerías, algoritmos y experimentos** (composición $\mathcal{A} \diamond \mathcal{L}$) para razonar sobre seguridad.
2. Construir **distinguidoras** y calcular probabilidades de salida para decidir si dos librerías son intercambiables ($L_0 \equiv L_1$) o no.

Ejercicio 1.

Considera los dos algoritmos $\mathcal{A}_1$, $\mathcal{A}_2$ y dos librerías $\mathcal{L}_1$, $\mathcal{L}_2$ con una interfaz común:

$\mathcal{A}_1$	$\mathcal{A}_2$	$\mathcal{L}_1$	$\mathcal{L}_2$
$r_1 := \text{RAND}(6)$ $r_2 := \text{RAND}(6)$ return $r_1 \stackrel{?}{=} r_2$	$r := \text{RAND}(6)$ return $r \stackrel{?}{\geq} 3$	$\underline{\text{RAND}(n):}$ $r \leftarrow \mathbb{Z}_n$ return r	$\underline{\text{RAND}(n):}$ return 0

Nuestro objetivo es demostrar formalmente que $\mathcal{L}_1$, $\mathcal{L}_2$ no son intercambiables. Para lograr esto:

- **Identifica un algoritmo distinguidor**: determina cuál de los dos programas, $\mathcal{A}_1$ o $\mathcal{A}_2$, tiene la capacidad de distinguir entre $\mathcal{L}_1$ y $\mathcal{L}_2$.
- **Demuestra la no intercambiabilidad**: usando el distinguidor, muestra que la definición de intercambiabilidad no se cumple, es decir,

$$\Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow \text{true}] \neq \Pr[\mathcal{A} \diamond \mathcal{L}_2 \Rightarrow \text{true}],$$

  donde $\mathcal{A}$ es el distinguidor escogido.

Solución detallada

Ambos algoritmos distinguen, pero basta con dar uno.
Opción 1 (más directa): tomamos $\mathcal{A}_1$.

- En $\mathcal{A}_1 \diamond \mathcal{L}_1$, las llamadas a RAND(6) devuelven r_1, r_2 independientes y uniformes en $\mathbb{Z}_6$. Entonces

$$\Pr[\mathcal{A}_1 \diamond \mathcal{L}_1 \Rightarrow \text{true}] = \Pr[r_1 = r_2] = \sum_{a \in \mathbb{Z}_6} \Pr[r_1 = a] \Pr[r_2 = a] = 6 \cdot \frac{1}{6} \cdot \frac{1}{6} = \frac{1}{6}.$$

- En $\mathcal{A}_1 \diamond \mathcal{L}_2$, siempre se tiene $r_1 = 0$ y $r_2 = 0$, así que

$$\Pr[\mathcal{A}_1 \diamond \mathcal{L}_2 \Rightarrow \text{true}] = 1.$$

Como $\frac{1}{6} \neq 1$, existe un algoritmo (por ejemplo $\mathcal{A}_1$) para el cual las probabilidades difieren, luego $\mathcal{L}_1$ y $\mathcal{L}_2$ **no** son intercambiables.

(Opcional) También funciona $\mathcal{A}_2$: en $\mathcal{L}_1$, $\Pr[r \geq 3] = 3/6 = 1/2$, mientras que en $\mathcal{L}_2$ siempre $r = 0$ y la probabilidad es 0.

Notas didácticas (para el alumno)

- **Estructura conceptual:** una librería induce una distribución sobre las salidas de $\mathcal{A}$; "intercambiables" significa mismas probabilidades para todo $\mathcal{A}$.
- **Estructura conceptual:** para demostrar *no* intercambiabilidad basta exhibir **un** distinguidor con probabilidades distintas.
- **Por qué este método:** aquí $\mathcal{L}_2$ es constante (devuelve siempre 0), así que conviene probar un evento que sea muy sensible a "constante vs aleatorio".
- **Por qué este método:** con $\mathcal{A}_1$ el evento $r_1 = r_2$ se calcula con una cuenta simple usando independencia.
- **Comprobaciones rápidas:** en $\mathcal{L}_2$ el resultado de $\mathcal{A}_1$ es determinista (true siempre).
- **Comprobaciones rápidas:** si r_1, r_2 son uniformes en $\mathbb{Z}_6$, la igualdad debería ocurrir con probabilidad $1/6$ ("una entre seis").
- **Errores comunes:** olvidar que r_1 y r_2 son **independientes** en $\mathcal{L}_1$.
- **Errores comunes:** confundir $\mathbb{Z}_6$ con $\{1, \ldots, 6\}$: aquí $\mathbb{Z}_6 = \{0, 1, 2, 3, 4, 5\}$.
- **Conexiones globales:** este patrón se repite en criptografía: distinguir "aleatorio real" vs "simulador" (PRG/PRF/RO, etc.).
- **Ampliación (opcional):** ventaja del distinguidor: $|\Pr[\mathcal{A} \diamond L_1] - \Pr[\mathcal{A} \diamond L_2]|$.

Ejercicio 2.

En cada item, se describe un par de librerías. Indica si $\mathcal{L}_0 \equiv \mathcal{L}_1$ o no. Si es así, muestra cómo asignan probabilidades idénticas a todos los resultados. Si no, describe un distinguidor exitoso.

Asume que ambas librerías usan el mismo valor de n. ¿Depende tu respuesta alguna vez de la elección de n?

En la parte (a), $\bar{x}$ denota el complemento bitwise de x. En la parte (d), $x \& y$ denota el AND bitwise de las dos cadenas:

(a)

$\mathcal{L}_0$
QUERY():
$x \leftarrow \{0,1\}^n$
return x

$\mathcal{L}_1$
QUERY():
$x \leftarrow \{0,1\}^n$
$y := \bar{x}$
return y

(b)

$\mathcal{L}_0$
QUERY():
$x \leftarrow \mathbb{Z}_n$
return x

$\mathcal{L}_1$
QUERY():
$x \leftarrow \mathbb{Z}_n$
$y := 2x \mod n$
return y

(c)

$\mathcal{L}_0$
QUERY($c \in \mathbb{Z}_n$):
if $c = 0$
return null
$x \leftarrow \mathbb{Z}_n$
return x

$\mathcal{L}_1$
QUERY($c \in \mathbb{Z}_n$):
if $c = 0$
return null
$x \leftarrow \mathbb{Z}_n$
$y := cx \mod n$
return y

(d)

$\mathcal{L}_0$
QUERY():
$x \leftarrow \{0,1\}^n$
$y \leftarrow \{0,1\}^n$
return $x \& y$

$\mathcal{L}_1$
QUERY():
$z \leftarrow \{0,1\}^n$
return z

Solución detallada

(a) Sí, $L_0 \equiv L_1$ para todo n. La transformación $x \mapsto \bar{x}$ es una biyección de $\{0,1\}^n$ en sí misma. Para cualquier $y \in \{0,1\}^n$,

$$\Pr[L_1 \Rightarrow y] = \Pr[\bar{x} = y] = \Pr[x = \bar{y}] = 2^{-n} = \Pr[L_0 \Rightarrow y].$$

(b) Depende de n. Sea $f(x) = 2x \bmod n$.

- Si $\gcd(2, n) = 1$ (equivalentemente, n es impar), entonces f es una permutación de $\mathbb{Z}_n$, y por tanto $f(x)$ es uniforme cuando x es uniforme. Luego $L_0 \equiv L_1$.
- Si n es par, $n = 2k$, entonces $2x \bmod n$ solo puede tomar valores pares $\{0, 2, 4, \ldots, 2k-2\}$. Por tanto $L_0 \not\equiv L_1$. Un distinguidor de una consulta es:

  $$\mathcal{A}: \ z := \text{QUERY}(); \ \text{return } (z \text{ es par}).$$

  Entonces $\Pr[\mathcal{A} \diamond L_0 \Rightarrow \text{true}] = 1/2$ y $\Pr[\mathcal{A} \diamond L_1 \Rightarrow \text{true}] = 1$.

(c) Depende de n y del valor de c. Si $c = 0$, ambas devuelven `null`. Para $c \neq 0$:

- Si $\gcd(c, n) = 1$, la función $x \mapsto cx \bmod n$ es una permutación de $\mathbb{Z}_n$. Luego la salida de L_1 es uniforme en $\mathbb{Z}_n$, igual que en L_0, y $L_0 \equiv L_1$ (para ese c).
- Si $d := \gcd(c, n) > 1$, entonces $cx \bmod n$ siempre es múltiplo de d. En concreto, la imagen es

  $$S = \{0, d, 2d, \ldots, (n/d - 1)d\} \subsetneq \mathbb{Z}_n,$$

  y L_1 es uniforme sobre S (cada valor de S tiene exactamente d preimágenes). Un distinguidor es:

  $$\mathcal{A}: \ z := \text{QUERY}(c); \ \text{return } (z \in S).$$

  Entonces $\Pr[\mathcal{A} \diamond L_1 \Rightarrow \text{true}] = 1$, mientras que

  $$\Pr[\mathcal{A} \diamond L_0 \Rightarrow \text{true}] = \Pr[z \in S] = \frac{|S|}{n} = \frac{n/d}{n} = \frac{1}{d}.$$

Conclusión práctica: si n es primo, entonces para todo $c \neq 0$ se tiene $\gcd(c, n) = 1$ y $L_0 \equiv L_1$; si n es compuesto, existe algún c con $\gcd(c, n) > 1$ y entonces **no** son intercambiables (porque el algoritmo puede elegir ese c).

(d) No, $L_0 \not\equiv L_1$ para todo $n \geq 1$. Considera el distinguidor

$$\mathcal{A} : \ s := \text{QUERY}(); \ \text{return } (s = 0^n).$$

En L_1, s es uniforme, luego

$$\Pr[\mathcal{A} \diamond L_1 \Rightarrow \text{true}] = \Pr[s = 0^n] = 2^{-n}.$$

En L_0, cada bit de $x\&y$ vale 0 con probabilidad 3/4 (solo es 1 si ambos bits son 1), y los bits son independientes, así que

$$\Pr[\mathcal{A} \diamond L_0 \Rightarrow \text{true}] = \Pr[x\&y = 0^n] = \left(\frac{3}{4}\right)^n.$$

Como $(3/4)^n \neq 2^{-n}$ para todo $n \geq 1$, las librerías no son intercambiables.

Resumen sobre dependencia en n: (a) no depende; (b) depende de la paridad de n; (c) depende de $\gcd(c, n)$ (y por tanto de si n es primo/compuesto); (d) no depende (siempre no intercambiables).

Notas didácticas (para el alumno)

- **Estructura conceptual:** $L_0 \equiv L_1$ significa "misma distribución de salida" (para toda secuencia de consultas permitidas).
- **Estructura conceptual:** una biyección sobre el espacio de salidas preserva la uniformidad (caso (a)).
- **Por qué este método:** en (b) y (c) el objeto clave es "multiplicar por una constante módulo n"; su comportamiento lo decide el gcd.
- **Por qué este método:** en (d) conviene elegir un evento fácil de calcular y con probabilidad claramente distinta (aquí $s = 0^n$).
- **Comprobaciones rápidas:** si $\gcd(c, n) = 1$, piensa "tiene inverso" $\Rightarrow$ es permutación $\Rightarrow$ uniforme.
- **Comprobaciones rápidas:** si $\gcd(c, n) > 1$, mira el soporte: cx mód n cae en un subconjunto (pista inmediata de distinguidor).
- **Errores comunes:** afirmar "uniforme" sin justificar que la función es permutación (o sin contar preimágenes).
- **Errores comunes:** en (d) suponer que $x\&y$ es uniforme: cada bit tiene sesgo ($\Pr[1] = 1/4$).
- **Conexiones globales:** estos argumentos son el núcleo de pruebas tipo "game hopping": reemplazar una distribución por otra usando simetrías/permutaciones.
- **Ampliación (opcional):** en (d) también puedes distinguir contando el peso de Hamming: en L_0 sigue $\text{Bin}(n, 1/4)$, en L_1 es $\text{Bin}(n, 1/2)$.

Ejercicio 3.

Demuestra que las siguientes librerías son intercambiables:

$\mathcal{L}_0$
QUERY$(m \in \{0,1\}^\lambda)$:
$x \leftarrow \{0,1\}^\lambda$
$y := x \oplus m$
return (x, y)

$\mathcal{L}_1$
QUERY$(m \in \{0,1\}^\lambda)$:
$y \leftarrow \{0,1\}^\lambda$
$x := y \oplus m$
return (x, y)

Nota que x e y se intercambian en las dos primeras líneas, pero no en la declaración de return.

Solución detallada

Fijamos un mensaje $m \in \{0,1\}^\lambda$ y calculamos la distribución de la salida (X, Y).

En $\mathcal{L}_0$, se elige $X \leftarrow \{0,1\}^\lambda$ uniforme y luego $Y = X \oplus m$. Por tanto, para cualesquiera $a, b \in \{0,1\}^\lambda$,

$$\Pr[\mathcal{L}_0(m) \Rightarrow (a,b)] = \begin{cases} 2^{-\lambda}, & \text{si } b = a \oplus m, \\ 0, & \text{en otro caso.} \end{cases}$$

En $\mathcal{L}_1$, se elige $Y \leftarrow \{0,1\}^\lambda$ uniforme y luego $X = Y \oplus m$. Así, para cualesquiera a, b,

$$\Pr[\mathcal{L}_1(m) \Rightarrow (a,b)] = \begin{cases} 2^{-\lambda}, & \text{si } a = b \oplus m, \\ 0, & \text{en otro caso.} \end{cases}$$

Pero las condiciones $b = a \oplus m$ y $a = b \oplus m$ son equivalentes (basta aplicar $\oplus m$ a ambos lados). En consecuencia, ambas librerías asignan la **misma** probabilidad a todo par (a, b), luego $\mathcal{L}_0 \equiv \mathcal{L}_1$.

Notas didácticas (para el alumno)

- **Estructura conceptual:** para probar $L_0 \equiv L_1$ basta comparar $\Pr[\text{salida} = (a,b)]$ para todo (a, b).
- **Estructura conceptual:** $x \mapsto x \oplus m$ es una biyección en $\{0,1\}^\lambda$.
- **Por qué este método:** la salida está definida por una restricción simple ($x \oplus y = m$), así que el cómputo directo es limpio.
- **Por qué este método:** evita "intuiciones" sobre one-time pad y deja una igualdad formal de distribuciones.
- **Comprobaciones rápidas:** en ambas librerías siempre se cumple $x \oplus y = m$.
- **Comprobaciones rápidas:** hay 2^λ pares posibles que satisfacen $x \oplus y = m$, todos equiprobables ($2^{-\lambda}$).
- **Errores comunes:** confundir "se intercambian líneas" con "se intercambia la salida": lo que importa es la distribución del `return`.
- **Errores comunes:** olvidar que $a = b \oplus m \Leftrightarrow b = a \oplus m$ (porque $\oplus m$ es su propia inversa).

- **Conexiones globales:** es el mismo tipo de simetría usada para probar seguridad de one-time pad y de máscaras XOR en reducciones.
- **Ampliación (opcional):** esta técnica es un "acoplamiento" explícito: emparejas ejecuciones con la misma variable aleatoria.

Ejercicio 4.

Muestra que las siguientes librerías no son intercambiables. Describe explícitamente un distinguidor y calcula sus probabilidades de salida cuando se vincula a ambas librerías:

$\mathcal{L}_0$
EAVESDROP$(m_0, m_1 \in \{0,1\}^\lambda)$:
$k \leftarrow \{0,1\}^\lambda$
$c := m_0 \oplus k$
return (k, c)

$\mathcal{L}_1$
EAVESDROP$(m_0, m_1 \in \{0,1\}^\lambda)$:
$k \leftarrow \{0,1\}^\lambda$
$c := m_1 \oplus k$
return (k, c)

Solución detallada

Construimos un distinguidor que elige mensajes conocidos y comprueba qué mensaje se usó para formar c.
Tomamos, por ejemplo, $m_0 = 0^\lambda$ y $m_1 = 1^\lambda$, y definimos:

$$\mathcal{A}: \ (k, c) \leftarrow \text{EAVESDROP}(0^\lambda, 1^\lambda); \ \text{return } (c = k).$$

- Si $\mathcal{A}$ se enlaza con $\mathcal{L}_0$, entonces $c = m_0 \oplus k = 0^\lambda \oplus k = k$, luego siempre $c = k$ y

$$\Pr[\mathcal{A} \diamond \mathcal{L}_0 \Rightarrow \text{true}] = 1.$$

- Si $\mathcal{A}$ se enlaza con $\mathcal{L}_1$, entonces $c = m_1 \oplus k = 1^\lambda \oplus k = \overline{k}$, y por tanto $c \neq k$ para todo k, luego

$$\Pr[\mathcal{A} \diamond \mathcal{L}_1 \Rightarrow \text{true}] = 0.$$

Como las probabilidades son distintas (de hecho, extremas), $\mathcal{L}_0$ y $\mathcal{L}_1$ no son intercambiables.

Notas didácticas (para el alumno)

- **Estructura conceptual:** el distinguidor puede *elegir* (m_0, m_1), así que conviene escogerlos con estructura simple.
- **Estructura conceptual:** si te dan (k, c), entonces $c \oplus k$ revela directamente el mensaje usado (aquí m_b).
- **Por qué este método:** elegir $m_0 = 0^\lambda$ hace que $c = k$ si se usa m_0; es una comprobación inmediata.
- **Por qué este método:** elegir $m_1 = 1^\lambda$ fuerza $c = \overline{k}$ si se usa m_1, separando totalmente los casos.
- **Comprobaciones rápidas:** en $\mathcal{L}_0$ siempre $c = k$; en $\mathcal{L}_1$ nunca $c = k$ (para $\lambda \geq 1$).
- **Comprobaciones rápidas:** alternativa equivalente: devolver `true` si $c \oplus k = 0^\lambda$.

- **Errores comunes:** pensar que k "debe ser secreto": el experimento devuelve k, por diseño.
- **Errores comunes:** escoger $m_0 = m_1$ (en ese caso sí serían indistinguibles para ese distinguidor).
- **Conexiones globales:** es el esquema de *juego IND* en su forma más básica: el atacante elige dos mensajes y ve un cifrado de uno.
- **Ampliación (opcional):** aquí la ventaja del distinguidor es 1 (distinción perfecta), un caso "demasiado inseguro".

Criptografía – Grado en Ingeniería de la Ciberseguridad

Hoja de problemas 2: OTP y PRG: distinguidoras y cierres

Escuela de Ingeniería Informática

Objetivos.

1. Practicar la definición de **intercambiabilidad / indistinguibilidad** en términos de librerías y distinguidoras, y el cálculo explícito de probabilidades.
2. Diseñar **distinguidoras simples** basadas en invariantes (paridad, conteo de unos, igualdad de mitades).
3. Dominar cierres básicos de PRG bajo transformaciones: permutaciones, proyecciones, ignorar bits, y reducciones.

Ejercicio 1.

Considera el siguiente esquema de cifrado. Soporta mensajes en claro de $\mathcal{M} = \{0,1\}^\lambda$ y textos cifrados de $\mathcal{C} = \{0,1\}^{2\lambda}$. Su espacio de claves es:

$$\mathcal{K} = \{k \in \{0,1,-\}^{2\lambda} \mid k \text{ contiene exactamente } \lambda \text{ caracteres “-”}\}.$$

Para cifrar un mensaje en claro m bajo la clave k, “rellenamos” los caracteres “-” en k usando los bits de m (en orden, de izquierda a derecha). Demuestra que el esquema no tiene intercambiabilidad de un solo uso, construyendo un programa que distinga las dos librerías relevantes de la definición de intercambiabilidad de un solo uso.

Ejemplo: a continuación se muestra un ejemplo de cifrado de $m = 1101100001$

$$\begin{aligned} k &= 1--0--11010-1-0-0--- \\ m &= \quad 1\,1 \quad 0\;\; 1 \qquad\quad 1 \quad 0 \quad 0 \quad 0\,0\,1 \end{aligned}$$

$$\Rightarrow \mathrm{Enc}(k,m) = 11100111010110000001.$$

Solución detallada

Consideramos las librerías de “intercambiabilidad de un solo uso” (una sola consulta) para este esquema Σ, que dependen de un bit $b \in \{0,1\}$:

$$\mathcal{L}^{\Sigma}_{\text{ot-int-}b}: \ \underline{\text{EAVESDROP}(m_0,m_1)} \text{ elige } k \leftarrow \mathcal{K} \text{ y devuelve } c := \mathrm{Enc}(k,m_b).$$

El atacante ve solo c.

Observación clave. Sea k una clave. Denotemos por $\#1(k)$ el número de unos entre los caracteres fijos de k (los que no son “-”). Al cifrar un mensaje m, los λ bits de m reemplazan exactamente los λ guiones, luego:

$$\#1(c) = \#1(k) + \#1(m).$$

Elegimos $m_0 := 0^\lambda$ y $m_1 := 1^\lambda$ y definimos el distinguidor:

$$\mathcal{A}: \ c \leftarrow \text{EAVESDROP}(0^\lambda, 1^\lambda); \ \text{return } (\#1(c) \geq \lambda).$$

Analizamos sus probabilidades.

- En $\mathcal{L}^{\Sigma}_{\text{ot-int-1}}$ se cifra $m_1 = 1^\lambda$, y por tanto

  $$\#1(c) = \#1(k) + \#1(1^\lambda) = \#1(k) + \lambda \geq \lambda,$$

  luego

  $$\Pr\left[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{ot-int-1}} \Rightarrow \text{true}\right] = 1.$$

- En $\mathcal{L}^{\Sigma}_{\text{ot-int-0}}$ se cifra $m_0 = 0^\lambda$, así que

  $$\#1(c) = \#1(k) + \#1(0^\lambda) = \#1(k) \leq \lambda.$$

  Se tiene $\#1(c) \geq \lambda$ *solo* si $\#1(k) = \lambda$, es decir, si todos los caracteres fijos de k son 1. Al muestrear k uniformemente de $\mathcal{K}$, los λ bits fijos son uniformes en $\{0,1\}^\lambda$, luego

  $$\Pr[\#1(k) = \lambda] = 2^{-\lambda}.$$

  Por tanto,

  $$\Pr\left[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{ot-int-0}} \Rightarrow \text{true}\right] = 2^{-\lambda}.$$

Como $1 \neq 2^{-\lambda}$, las librerías no son intercambiables, y por tanto el esquema **no** tiene intercambiabilidad de un solo uso. Además, la ventaja del distinguidor es

$$\text{Adv}(\mathcal{A}) = \left|1 - 2^{-\lambda}\right| = 1 - 2^{-\lambda}.$$

Notas didácticas (para el alumno)

- **Estructura conceptual:** para romper "un solo uso" basta **una** consulta a EAVESDROP con dos mensajes elegidos.
- **Estructura conceptual:** busca un *invariante* del cifrado que cambie mucho entre m_0 y m_1 (aquí, # de unos).
- **Por qué este método:** el esquema "inserta" los bits del mensaje sin difuminarlos; el conteo de unos se conserva de forma lineal.
- **Por qué este método:** elegir 0^λ y 1^λ maximiza la separación: $\#1(m_1) - \#1(m_0) = \lambda$.
- **Comprobaciones rápidas:** verifica mentalmente $\#1(c) = \#1(\text{fijos de } k) + \#1(m)$.
- **Comprobaciones rápidas:** en el mundo $b = 1$, siempre se cumple $\#1(c) \geq \lambda$.
- **Errores comunes:** olvidar que k tiene exactamente λ guiones, por lo que se usan *todos* los bits de m.
- **Errores comunes:** confundir "clave uniforme" con "posiciones uniformes": la probabilidad crítica aquí es que los bits fijos sean todos 1.

- **Conexiones globales:** este es un ejemplo típico de fuga de información por *estadísticas* del cifrado (no hay difusión).
- **Ampliación (opcional):** si cambias la regla a "rellenar con $m \oplus r$" para un r aleatorio, el ataque ya no funciona igual: aparece el patrón OTP.

Ejercicio 2.

Considere una función $G\colon \{0,1\}^n \to \{0,1\}^{n+1}$ definida como

$$G(s) = s \| (s_1 \oplus s_2 \oplus \cdots \oplus s_n).$$

Es decir, G concatena la semilla s con el XOR de todos sus bits. Determine si G es un generador pseudoaleatorio seguro.

Solución detallada

No es un PRG seguro. Para verlo, consideramos las librerías estándar:

$$\mathcal{L}^{G}_{\text{prg}} : \underline{\text{QUERY}()}\ s \leftarrow \{0,1\}^n;\ \text{return } G(s), \qquad \mathcal{L}^{\$}_{\text{prg}} : \underline{\text{QUERY}()}\ r \leftarrow \{0,1\}^{n+1};\ \text{return } r.$$

Construimos el distinguidor:

$$\mathcal{A} :\ r_1 \| \cdots \| r_{n+1} \leftarrow \text{QUERY}();\ \text{return } (r_{n+1} = r_1 \oplus \cdots \oplus r_n).$$

- En $\mathcal{L}^{G}_{\text{prg}}$, por definición $r_{n+1} = r_1 \oplus \cdots \oplus r_n$, luego

$$\Pr\left[\mathcal{A} \diamond \mathcal{L}^{G}_{\text{prg}} \Rightarrow \text{true}\right] = 1.$$

- En $\mathcal{L}^{\$}_{\text{prg}}$, los bits $r_1, \ldots, r_{n+1}$ son uniformes e independientes. Fijados $r_1, \ldots, r_n$, el bit r_{n+1} coincide con su XOR con probabilidad $1/2$, luego

$$\Pr\left[\mathcal{A} \diamond \mathcal{L}^{\$}_{\text{prg}} \Rightarrow \text{true}\right] = \frac{1}{2}.$$

Por tanto, la ventaja es

$$\text{Adv}(\mathcal{A}) = \left|1 - \frac{1}{2}\right| = \frac{1}{2},$$

y G no es un generador pseudoaleatorio seguro.

Notas didácticas (para el alumno)

- **Estructura conceptual:** un PRG seguro debe hacer que la salida sea indistinguible de uniforme para cualquier algoritmo eficiente.
- **Estructura conceptual:** una *relación determinista* entre bits de la salida suele dar un distinguidor inmediato.
- **Por qué este método:** el último bit está completamente determinado por los primeros n bits (paridad).
- **Por qué este método:** el test "ès consistente la paridad?" es computable en tiempo lineal.

- **Comprobaciones rápidas:** en el mundo real, el predicado se cumple siempre (probabilidad 1).
- **Comprobaciones rápidas:** en el mundo aleatorio, un bit independiente coincide con un valor fijo con probabilidad 1/2.
- **Errores comunes:** olvidar que basta **una** propiedad con probabilidad distinta (no hace falta "recuperar la semilla").
- **Errores comunes:** confundir "$n+1$ bits parecen aleatorios" con "son aleatorios": una ecuación lineal ya rompe.
- **Conexiones globales:** muchas pruebas de inseguridad de PRG/PRF se basan en invariantes lineales o algebraicos.
- **Ampliación (opcional):** si en vez de XOR pones un bit verdaderamente aleatorio extra, la relación desaparece (pero ya no es determinista).

Ejercicio 3.

Sea $G\colon \{0,1\}^\lambda \to \{0,1\}^{3\lambda}$ un PRG seguro que triplica la longitud. Para cada función a continuación, indique si también es un PRG seguro. Si la función es un PRG seguro, dé una prueba. Si no, describa un distinguidor exitoso y calcule explícitamente su ventaja. Cuando escribimos $a\|b\|c := G(s)$, cada uno de a, b, c tiene longitud λ.

1. $H(s) : x\|y\|z := G(s)$, return $G(x)\|G(z)$.
2. $H(s) : x\|y\|z := G(s)$, return $x\|y$.
3. $H(s) : x := G(s)$, $y := G(s)$, return $x\|y$.

Solución detallada

(1) Sí, es un PRG seguro. Considera una ejecución real de H: se toma $s \leftarrow \{0,1\}^\lambda$, se calcula $x\|y\|z = G(s)$ y se devuelve $G(x)\|G(z)$.
Hacemos un argumento por híbridos:
Híbrido 0 (real): $x\|y\|z := G(s)$; salida $G(x)\|G(z)$.
Híbrido 1: reemplazamos $G(s)$ por $r \leftarrow \{0,1\}^{3\lambda}$. Por seguridad de G, el tríple (x, y, z) es indistinguible de uniforme, así que H0 y H1 son indistinguibles para todo distinguidor eficiente.
En H1, x y z son uniformes e independientes en $\{0,1\}^\lambda$.
Híbrido 2: en H1, reemplazamos $G(x)$ por $u \leftarrow \{0,1\}^{3\lambda}$. Esto es válido por seguridad de G (usamos un distinguidor que tiene acceso a x como semilla y a $G(z)$ como "ruido" adicional). La pérdida de ventaja es a lo sumo $\mathrm{negl}(\lambda)$.
Híbrido 3: en H2, reemplazamos $G(z)$ por $v \leftarrow \{0,1\}^{3\lambda}$, con otra pérdida a lo sumo $\mathrm{negl}(\lambda)$.
En H3, la salida es $u\|v$ con u, v uniformes e independientes, o sea uniforme en $\{0,1\}^{6\lambda}$. Por tanto H es un PRG seguro (con ventaja acotada por la suma de negligibles).

(2) Sí, es un PRG seguro. Si hubiera un distinguidor $\mathcal{A}$ que distingue $x\|y$ de uniforme, construimos un distinguidor $\mathcal{B}$ contra G:

$$\mathcal{B}:\ t \leftarrow \mathrm{QUERY}();\ \text{parsea } t = x\|y\|z \text{ (con } |x| = |y| = |z| = \lambda);\ \text{return } \mathcal{A}(x\|y).$$

Si $t = G(s)$, entonces $x\|y$ está distribuido como $H(s)$. Si t es uniforme de longitud 3λ, entonces $x\|y$ es uniforme de longitud 2λ. Luego $\mathrm{Adv}(\mathcal{A}) = \mathrm{Adv}(\mathcal{B})$, y como G es seguro, $\mathcal{A}$ no puede existir. Concluimos que H es un PRG seguro.

(3) No es un PRG seguro. Definimos el distinguidor:

$$\mathcal{A}: \; x\|y \leftarrow \mathrm{QUERY}(); \; \text{return } (x = y).$$

En el mundo real, $x = G(s)$ y $y = G(s)$, por determinismo, así que

$$\Pr\left[\mathcal{A} \diamond \mathcal{L}^{H}_{\mathrm{prg}} \Rightarrow \mathrm{true}\right] = 1.$$

En el mundo aleatorio, x e y son uniformes e independientes en $\{0,1\}^{3\lambda}$, luego

$$\Pr\left[\mathcal{A} \diamond \mathcal{L}^{\$}_{\mathrm{prg}} \Rightarrow \mathrm{true}\right] = \Pr[x = y] = 2^{-3\lambda}.$$

Por tanto,

$$\mathrm{Adv}(\mathcal{A}) = \left|1 - 2^{-3\lambda}\right| = 1 - 2^{-3\lambda},$$

y H no es seguro.

Notas didácticas (para el alumno)

- **Estructura conceptual:** para probar seguridad, sustituye partes "pseudoaleatorias" por uniformes en una cadena de híbridos.
- **Estructura conceptual:** tomar un subconjunto de bits de una cadena pseudoaleatoria preserva pseudorandomness (reducción directa).
- **Por qué este método:** en (1) y (2) hay una forma natural de reducir a la seguridad de G (misma definición, distinta post-procesado).
- **Por qué este método:** en (3) hay una propiedad determinista ($x = y$) que da un distinguidor de una línea.
- **Comprobaciones rápidas:** si algo es uniforme en 3λ, entonces sus bloques de λ bits son uniformes e independientes.
- **Comprobaciones rápidas:** en (3), $\Pr[x = y]$ para dos uniformes independientes de longitud 3λ es $2^{-3\lambda}$.
- **Errores comunes:** suponer "si cada bloque parece aleatorio, entonces todo lo es" (ojo con correlaciones).
- **Errores comunes:** olvidar que "G es determinista" implica igualdades exactas que un aleatorio no cumple.
- **Conexiones globales:** este tipo de cierres se usa para construir PRG largos, PRF, y para híbridos en pruebas IND.
- **Ampliación (opcional):** en (1) la ventaja final queda acotada por la suma de ventajas de las sustituciones (regla del triángulo).

Ejercicio 4.

Sea H un generador pseudoaleatorio seguro. En cada uno de los siguientes casos, determine si G es un generador pseudoaleatorio seguro.

1. $G(s) = H(\bar{s})$ donde $\bar{x}$ denota el complemento bit a bit de x.
2. $G(s) = \overline{H(s)}$.
3. $G(s_L \| s_R) = H(s_L)$.
4. $G(s_1 \| \cdots \| s_n) = H(s_1 \| \cdots \| s_{n-1})$, donde $s_j \in \{0,1\}$ son bits.

Nota: Podéis utilizar las librerías intercambiables que aparecen en la Hoja 1.

Solución detallada

En los cuatro casos, G sigue siendo un PRG seguro.

(1) $G(s) = H(\bar{s})$ **es seguro.** La aplicación $s \mapsto \bar{s}$ es una permutación de $\{0,1\}^\lambda$. Si s es uniforme, entonces $\bar{s}$ también lo es. Por tanto, la distribución de $H(\bar{s})$ es exactamente la misma que la de $H(s)$. Luego G es tan seguro como H.

(2) $G(s) = \overline{H(s)}$ **es seguro.** Complementar bits es una permutación de $\{0,1\}^\ell$ (si H expande a ℓ bits). Si existiera un distinguidor para $\overline{H(s)}$, podríamos convertirlo en uno para $H(s)$ simplemente complementando su entrada antes de ejecutarlo. Así, la ventaja se preserva y, como H es seguro, G también.

(3) $G(s_L \| s_R) = H(s_L)$ **es seguro.** Si $s_L \| s_R$ es uniforme en $\{0,1\}^{2\lambda}$, entonces s_L es uniforme en $\{0,1\}^\lambda$, y s_R se ignora por completo. La salida de G tiene exactamente la misma distribución que H sobre una semilla uniforme, luego G es seguro.

(4) $G(s_1 \| \cdots \| s_n) = H(s_1 \| \cdots \| s_{n-1})$ **es seguro.** Si $s = s_1 \| \cdots \| s_n$ es uniforme, entonces el prefijo $s' := s_1 \| \cdots \| s_{n-1}$ es uniforme en $\{0,1\}^{n-1}$, y el último bit se ignora. Por tanto, la salida se distribuye como $H(s')$ con semilla uniforme, y G es seguro.

Notas didácticas (para el alumno)

- **Estructura conceptual:** una permutación de semillas preserva "semilla uniforme" (caso (1)).
- **Estructura conceptual:** una permutación de salidas preserva "salida uniforme" y, por reducción, preserva pseudorandomness (caso (2)).
- **Por qué este método:** en (1)–(2) el cambio es solo una transformación biyectiva; no crea estructura detectable.
- **Por qué este método:** en (3)–(4) se ignoran bits de la semilla; con semilla uniforme, el resto sigue uniforme.
- **Comprobaciones rápidas:** si S es uniforme, entonces $\bar{S}$ también lo es.
- **Comprobaciones rápidas:** si R es uniforme, entonces $\overline{R}$ también lo es.
- **Errores comunes:** pensar que "ignorar bits" debilita por sí mismo: con semilla uniforme no introduce sesgo en el prefijo.
- **Errores comunes:** olvidar que los distinguidoras pueden invertir permutaciones eficientemente (por eso se usan reducciones directas).

- **Conexiones globales:** estas ideas aparecen en cierres de PRF/PRP y en híbridos: re-etiquetar o re-codificar no cambia seguridad.
- **Ampliación (opcional):** si en (3) la semilla no fuera uniforme (p.ej. elegida adversarialmente), la conclusión podría fallar: aquí la uniformidad es esencial.

Ejercicio 5.

Sea H un generador pseudoaleatorio seguro. Considera las siguientes funciones:

1. $G'(s) = G(0^{|s|} \| s)$ donde $G(s_L \| s_R) = H(s_L)$.
2. $G'(s) = G(s) \| G(s+1)$ donde $G(s_1 \| \cdots \| s_n) = H(s_1 \| \cdots \| s_{n-1})$.

Observa que G es un generador pseudoaleatorio seguro. Demuestra que G' **no** es seguro.

Solución detallada

(1) $G'(s) = G(0^{|s|} \| s)$ **no es seguro.** Como $G(s_L \| s_R) = H(s_L)$, se tiene

$$G'(s) = G(0^{|s|} \| s) = H(0^{|s|}),$$

que es una cadena *constante* (no depende de s).
Definimos el distinguidor:

$$\mathcal{A}: \ x \leftarrow \text{QUERY}(); \ \text{return } (x = H(0^{|s|})).$$

En el mundo real, siempre se cumple $x = H(0^{|s|})$, luego la probabilidad es 1. En el mundo aleatorio, x es uniforme de longitud ℓ (la longitud de salida), y coincide con un valor fijo con probabilidad $2^{-\ell}$. Por tanto,

$$\text{Adv}(\mathcal{A}) = 1 - 2^{-\ell}.$$

(2) $G'(s) = G(s) \| G(s+1)$ **no es seguro.** Aquí $G(s) = H(\text{pref}(s))$, donde $\text{pref}(s)$ es el prefijo de longitud $n-1$.
Si el último bit de s es 0, entonces $s+1$ cambia solo ese bit a 1 (sin acarreo), y el prefijo se mantiene:

$$\text{pref}(s) = \text{pref}(s+1).$$

En ese caso, $G(s) = G(s+1)$ y por tanto las dos mitades de $G'(s)$ son iguales.
Tomamos el distinguidor:

$$\mathcal{A}: \ x \| y \leftarrow \text{QUERY}(); \ \text{return } (x = y).$$

En el mundo real, la semilla s es uniforme en $\{0,1\}^n$ y ocurre $s_n = 0$ con probabilidad $1/2$. En ese evento, $x = y$ con probabilidad 1; fuera de ese evento, $x = y$ como mucho con probabilidad despreciable. En particular,

$$\Pr[\mathcal{A} \text{ devuelve true en real}] \geq \frac{1}{2}.$$

En el mundo aleatorio, x e y son uniformes e independientes (cada uno de longitud ℓ), y

$$\Pr[x = y] = 2^{-\ell}.$$

Luego, una cota explícita para la ventaja es

$$\text{Adv}(\mathcal{A}) \geq \left| \frac{1}{2} - 2^{-\ell} \right|.$$

Esto es no despreciable, así que G' no es seguro.

Notas didácticas (para el alumno)

- **Estructura conceptual:** un PRG seguro no puede producir salidas con estructura determinista fácil de testear.
- **Estructura conceptual:** concatenar dos salidas puede introducir correlaciones (igualdad de mitades, repeticiones).
- **Por qué este método:** en (1) G' ignora completamente s en la parte relevante, dando una salida constante.
- **Por qué este método:** en (2) hay muchas semillas para las que $G(s) = G(s+1)$ (cuando no hay acarreo).
- **Comprobaciones rápidas:** "salida constante" se distingue comparando con ese valor fijo.
- **Comprobaciones rápidas:** para dos cadenas uniformes independientes, $\Pr[x = y] = 2^{-|x|}$.
- **Errores comunes:** asumir que "si G es seguro, entonces $G(s)\|G(s+1)$ lo es": s y $s+1$ no son independientes.
- **Errores comunes:** olvidar el efecto de los acarreos: basta con usar el evento simple "último bit $= 0$" para obtener probabilidad $1/2$.
- **Conexiones globales:** este es un patrón clásico: *reusar* (o correlacionar) semillas rompe propiedades pseudoaleatorias.
- **Ampliación (opcional):** si en vez de $s+1$ usas una segunda semilla independiente, entonces el ataque por igualdad de mitades desaparece.

Ejercicio 6.

Sean G_1 y G_2 funciones deterministas, cada una aceptando entradas de longitud λ y produciendo salidas de longitud 3λ. Define

$$H(s_1\|s_2) = G_1(s_1) \oplus G_2(s_2).$$

Demuestra que si cualquiera de G_1 o G_2 (o ambos) es un PRG seguro, entonces también lo es H.

Solución detallada

Basta probar el caso en que G_1 es un PRG seguro (el caso de G_2 es análogo, e incluso si ambos lo son la conclusión sigue valiendo).
Considera un distinguidor arbitrario $\mathcal{A}$ que intenta distinguir la salida de H de una cadena uniforme de longitud 3λ.
Construimos un distinguidor $\mathcal{B}$ contra G_1:

Distinguidor $\mathcal{B}$:
Entrada: acceso a un oráculo QUERY() que devuelve o bien $G_1(s_1)$ con $s_1 \leftarrow \{0,1\}^\lambda$, o bien $r \leftarrow \{0,1\}^{3\lambda}$.
1) Elige $s_2 \leftarrow \{0,1\}^\lambda$ y calcula $t := G_2(s_2)$.
2) Pide $u \leftarrow$ QUERY().
3) Devuelve lo que devuelva $\mathcal{A}(u \oplus t)$.

Si el oráculo de $\mathcal{B}$ está en el mundo real, entonces $u = G_1(s_1)$ y la entrada a $\mathcal{A}$ es

$$u \oplus t = G_1(s_1) \oplus G_2(s_2) = H(s_1 \| s_2),$$

o sea, la distribución real de H.
Si el oráculo está en el mundo aleatorio, entonces u es uniforme en $\{0,1\}^{3\lambda}$. Como XOR con una constante t es una permutación, $u \oplus t$ sigue siendo uniforme. Por tanto, $\mathcal{A}$ ve exactamente el mundo aleatorio.
Concluimos que $\mathrm{Adv}(\mathcal{A}) = \mathrm{Adv}(\mathcal{B})$, y como G_1 es seguro, la ventaja es despreciable. Luego H es un PRG seguro.

Notas didácticas (para el alumno)

- **Estructura conceptual:** la prueba es una reducción: si alguien rompe H, entonces rompe G_1 o G_2.
- **Estructura conceptual:** XOR con una cadena fija es una permutación del espacio $\{0,1\}^{3\lambda}$.
- **Por qué este método:** si $G_1(s_1)$ parece aleatorio, entonces $G_1(s_1) \oplus t$ también parece aleatorio ("one-time pad" con máscara fija).
- **Por qué este método:** $G_2(s_2)$ actúa como máscara eficiente que podemos computar dentro de la reducción.
- **Comprobaciones rápidas:** si U es uniforme, entonces $U \oplus t$ es uniforme para cualquier t fijo.
- **Comprobaciones rápidas:** si U es indistinguible de uniforme, también lo es $U \oplus t$ (mismo argumento de permutación).
- **Errores comunes:** pensar que XOR "suma entropía": si ambos fueran malos, el XOR puede seguir siendo malo; aquí la hipótesis es "al menos uno es bueno".
- **Errores comunes:** olvidar que la reducción debe ser eficiente y usar solo una llamada a su oráculo (aquí, una).
- **Conexiones globales:** esto es el mismo principio que enmascaramiento OTP y en híbridos: "aleatorio $\oplus$ cualquier cosa = aleatorio".
- **Ampliación (opcional):** si G_1 y G_2 fueran PRF y el XOR se hiciera punto a punto, el resultado análogo requiere más cuidado (dependencias).

Criptografía – Grado en Ingeniería de la Ciberseguridad

Hoja de problemas 3: Modos de operación y seguridad CPA/CCA

Escuela de Ingeniería Informática

Objetivos.

1. Entender **propagación de errores** en CBC/OFB/CTR cuando se pierde un bloque del cifrado.
2. Practicar ataques de **no seguridad CPA** y **no seguridad CCA** construyendo distinguidoras con ventaja explícita.
3. Reconocer cúando un modo es **maleable** y cómo explotar esa maleabilidad en un juego CCA.

Ejercicio 1.

Considere un mensaje cifrado $c = (c_0, c_1, c_2, \ldots)$ por bloques. Suponga que el receptor recibe c de manera incompleta, ya que le falta c_i para algún $i \geq 1$. Comente el efecto que tiene este fallo en el descifrado para cada uno de los siguientes modos de operación: CBC, OFB y CTR.

Solución detallada

Supondremos que el receptor conoce en qué posición falta el bloque ("falta c_i"), pero **no** recibe un bloque de relleno: el flujo de bloques queda *desplazado* a partir de i.

CBC. El descifrado es

$$m_j = F_k^{-1}(c_j) \oplus c_{j-1} \quad (j \geq 1), \qquad c_0 = \mathrm{IV}.$$

Si falta c_i:

- Los bloques anteriores $m_1, \ldots, m_{i-1}$ se recuperan sin problemas (usan solo bloques anteriores).
- El bloque m_i **no se puede** recuperar, porque requiere c_i.
- El bloque m_{i+1} queda **corrupto** (o no recuperable), porque requiere c_i en el XOR: $m_{i+1} = F_k^{-1}(c_{i+1}) \oplus c_i$.
- A partir de m_{i+2}, el descifrado vuelve a ser correcto (si los bloques vuelven a estar alineados), porque ya no depende de c_i.

Conclusión: en CBC la pérdida de un bloque afecta **a lo sumo a dos bloques** de texto en claro: m_i (se pierde) y m_{i+1} (se corrompe).

OFB. En OFB se genera un flujo $r_1, r_2, \ldots$ a partir del IV:

$$r_1 = F_k(\mathrm{IV}), \quad r_j = F_k(r_{j-1}), \qquad c_j = m_j \oplus r_j, \quad m_j = c_j \oplus r_j.$$

Si falta un bloque c_i y el receptor no puede "insertar" un marcador para mantener la alineación, entonces a partir de esa posición se descuadra el XOR: el bloque que llega en la posición i en realidad es c_{i+1}, y se descifra como $m'_i = c_{i+1} \oplus r_i$, etc. Por tanto:

- $m_1, \ldots, m_{i-1}$ se recuperan correctamente.
- m_i se pierde (no hay c_i).
- Los bloques posteriores quedan **ininteligibles** hasta que exista un mecanismo de resincronización (por ejemplo, numeración/segmentación).

CTR. En CTR el flujo depende de un contador:

$$r_j = F_k(\mathrm{IV} + j), \qquad c_j = m_j \oplus r_j, \qquad m_j = c_j \oplus r_j.$$

La situación es análoga a OFB: si falta c_i y el receptor no resincroniza el contador con el índice correcto, a partir de ese punto los bloques se descifran con el contador desplazado y el resultado es **ininteligible**. En resumen:

- $m_1, \ldots, m_{i-1}$ correctos.
- m_i se pierde.
- $m_{i+1}, m_{i+2}, \ldots$ quedan mal si no se recupera la alineación del contador.

Notas didácticas (para el alumno)

- **Estructura conceptual:** "pérdida de un bloque" puede causar (i) pérdida local o (ii) pérdida de sincronización.
- **Estructura conceptual:** CBC encadena *cifrados*, OFB/CTR encadenan (o indexan) *flujo de clave*.
- **Por qué este método:** en CBC basta mirar la fórmula $m_j = F_k^{-1}(c_j) \oplus c_{j-1}$ para ver qué depende de c_i.
- **Por qué este método:** en OFB/CTR, si el XOR se aplica con el flujo equivocado, todo queda desplazado.
- **Comprobaciones rápidas:** en CBC, c_i aparece solo en m_i (dentro de F^{-1}) y en m_{i+1} (como XOR).
- **Comprobaciones rápidas:** en OFB/CTR cada bloque se descifra como $m_i = c_i \oplus r_i$ (si r_i está bien alineado).
- **Errores comunes:** confundir "bit flip" (propaga distinto) con "bloque perdido" (desincroniza).
- **Errores comunes:** olvidar que OFB/CTR son *cifrados de flujo*: necesitan sincronización en el índice.
- **Conexiones globales:** en protocolos reales, CTR/OFB suelen incluir contador/-nonce y segmentación para poder resincronizar.
- **Ampliación (opcional):** si el canal conserva posiciones (p.ej. con números de bloque), OFB/CTR pueden recuperar m_{i+1}, m_{i+2}, etc. (solo se pierde m_i).

Ejercicio 2.

Supongamos que en lugar de aplicar el modo CBC a un cifrado de bloques, lo aplicamos a un one-time pad. En otras palabras, reemplazamos cada ocurrencia de $F(k, \star)$ con $k \oplus \star$ en el código para el cifrado CBC. Demuestre que el resultado no tiene seguridad CPA. Describa un distinguidor y calcule su ventaja.

Solución detallada

Sea la longitud de bloque λ y una clave fija $k \in \{0,1\}^\lambda$. El "CBC con OTP" queda:

$$c_0 \leftarrow \{0,1\}^\lambda, \qquad c_i := k \oplus m_i \oplus c_{i-1} \quad (i = 1, \ldots, \ell).$$

(Es decir, se usa el *mismo* pad k en todos los bloques.)
Consideramos las librerías CPA habituales $\mathcal{L}^{\Sigma}_{\text{cpa-}b}$ con una consulta `EAVESDROP` que cifra m_b.

Elegimos mensajes de dos bloques:

$$M_0 := 0^\lambda \| 0^\lambda, \qquad M_1 := 1^\lambda \| 0^\lambda.$$

Distinguidor:

$$\mathcal{A}: \ c_0 \| c_1 \| c_2 \leftarrow \text{EAVESDROP}(M_0, M_1); \ \text{return } (c_2 = c_0).$$

Analizamos:

- Si $b = 0$ (se cifra M_0), entonces

  $$c_1 = k \oplus 0^\lambda \oplus c_0 = k \oplus c_0, \qquad c_2 = k \oplus 0^\lambda \oplus c_1 = k \oplus (k \oplus c_0) = c_0.$$

  Por tanto, $\Pr\left[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{cpa-0}} \Rightarrow \text{true}\right] = 1$.

- Si $b = 1$ (se cifra M_1), entonces

  $$c_1 = k \oplus 1^\lambda \oplus c_0, \qquad c_2 = k \oplus 0^\lambda \oplus c_1 = k \oplus (k \oplus 1^\lambda \oplus c_0) = 1^\lambda \oplus c_0 \neq c_0.$$

  Por tanto, $\Pr\left[\mathcal{A} \diamond \mathcal{L}^{\Sigma}_{\text{cpa-1}} \Rightarrow \text{true}\right] = 0$.

Luego la ventaja es

$$\text{Adv}(\mathcal{A}) = |1 - 0| = 1,$$

y el esquema **no** tiene seguridad CPA.

Notas didácticas (para el alumno)

- **Estructura conceptual:** CPA pide que un cifrado de m_0 sea indistinguible de un cifrado de m_1 (con mismo número de bloques).
- **Estructura conceptual:** "usar el mismo pad en muchos bloques" introduce relaciones lineales entre bloques del cifrado.
- **Por qué este método:** elegimos M_0, M_1 para forzar una identidad exacta ($c_2 = c_0$) en un mundo y su negación en el otro.
- **Por qué este método:** el test $c_2 = c_0$ es de coste $O(\lambda)$ y da ventaja 1.

- **Comprobaciones rápidas:** calcula c_2 sustituyendo recursivamente: $c_2 = k \oplus m_2 \oplus (k \oplus m_1 \oplus c_0) = m_1 \oplus m_2 \oplus c_0$.
- **Comprobaciones rápidas:** en un cifrado sano, igualdades exactas entre bloques (sin motivo) deben ocurrir con probabilidad despreciable.
- **Errores comunes:** pensar "OTP es seguro" sin notar que aquí el pad **se reutiliza** por bloque.
- **Errores comunes:** mezclar el papel de c_0 (IV aleatorio) con la clave k (fija).
- **Conexiones globales:** este ataque es el mismo principio que rompe "two-time pad": reutilizar XOR-key filtra XORs de mensajes.
- **Ampliación (opcional):** también puedes recuperar k si conoces un bloque m_1: $k = c_1 \oplus m_1 \oplus c_0$.

Ejercicio 3.

A continuación se muestran dos modos de cifrado de bloques aplicados a un PRP F con longitud de bloque λ. Para cada modo:

- Describe el procedimiento de descifrado correspondiente.
- Demuestra que el modo no tiene seguridad CPA: describe un distinguidor y calcula su ventaja.

(a)
$\underline{\mathrm{Enc}(k, m_1\| \cdots \|m_\ell)}$:
$r_0 \leftarrow \{0,1\}^\lambda$
$c_0 := r_0$
for $i := 1$ to ℓ
  $r_i := F(k, m_i)$
  $c_i := r_i \oplus r_{i-1}$
return $c_0\| \cdots \|c_\ell$

(b)
$\underline{\mathrm{Enc}(k, m_1\| \cdots \|m_\ell)}$:
$c_0 \leftarrow \{0,1\}^\lambda$
$m_0 := c_0$
for $i := 1$ to ℓ
  $c_i := F(k, m_i) \oplus m_{i-1}$
return $c_0\| \cdots \|c_\ell$

Solución detallada

(a) Descifrado. Como $c_0 = r_0$, podemos recuperar iterativamente r_i:

$$r_0 := c_0, \qquad r_i := c_i \oplus r_{i-1} \ (i \geq 1),$$

y luego

$$m_i := F^{-1}(k, r_i) \quad (i = 1, \ldots, \ell).$$

(a) No seguridad CPA. Tomamos mensajes de dos bloques:

$$M_0 := 0^\lambda \| 0^\lambda, \qquad M_1 := 0^\lambda \| 1^\lambda.$$

Distinguidor:

$$\mathcal{A}: \ c_0\|c_1\|c_2 \leftarrow \mathrm{EAVESDROP}(M_0, M_1); \ \text{return } (c_2 = 0^\lambda).$$

Si se cifra M_0, entonces $r_1 = F(k, 0^\lambda)$ y $r_2 = F(k, 0^\lambda) = r_1$, luego $c_2 = r_2 \oplus r_1 = 0^\lambda$ con probabilidad 1. Si se cifra M_1, entonces $r_1 = F(k, 0^\lambda)$ y $r_2 = F(k, 1^\lambda)$, y como $F(k, \cdot)$ es una permutación se tiene $r_2 \neq r_1$, por tanto $c_2 = r_2 \oplus r_1 \neq 0^\lambda$ con probabilidad 1. Así,

$$\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{cpa-0}} \Rightarrow \text{true}] = 1, \qquad \Pr[\mathcal{A} \diamond \mathcal{L}_{\text{cpa-1}} \Rightarrow \text{true}] = 0,$$

y $\mathrm{Adv}(\mathcal{A}) = 1$.

(b) Descifrado. Aquí $m_0 = c_0$ es conocido. Para $i \geq 1$:

$$c_i = F(k, m_i) \oplus m_{i-1} \implies F(k, m_i) = c_i \oplus m_{i-1} \implies m_i = F^{-1}(k, c_i \oplus m_{i-1}).$$

Por tanto:

$$m_0 := c_0, \qquad m_i := F^{-1}(k, c_i \oplus m_{i-1}) \ (i = 1, \ldots, \ell).$$

(b) No seguridad CPA. Usamos de nuevo mensajes de dos bloques:

$$M_0 := 0^\lambda \| 0^\lambda, \qquad M_1 := 0^\lambda \| 1^\lambda.$$

Sea $m_1 = 0^\lambda$ en ambos. Al cifrar:

$$c_1 = F(k, m_1) \oplus m_0 = F(k, 0^\lambda) \oplus c_0,$$

así que $F(k, 0^\lambda) = c_1 \oplus c_0$ es *observable* desde el cifrado. Además,

$$c_2 = F(k, m_2) \oplus m_1 = F(k, m_2) \oplus 0^\lambda = F(k, m_2).$$

Por tanto, el distinguidor:

$$\mathcal{A}: \ c_0 \| c_1 \| c_2 \leftarrow \mathrm{EAVESDROP}(M_0, M_1); \ \text{return} \ (c_2 = c_1 \oplus c_0)$$

satisface:

- Si $b = 0$, entonces $m_2 = 0^\lambda$ y $c_2 = F(k, 0^\lambda) = c_1 \oplus c_0$, luego probabilidad 1.
- Si $b = 1$, entonces $m_2 = 1^\lambda$ y $c_2 = F(k, 1^\lambda) \neq F(k, 0^\lambda) = c_1 \oplus c_0$, luego probabilidad 0.

Así, $\mathrm{Adv}(\mathcal{A}) = 1$ y (b) no es CPA-seguro.

Notas didácticas (para el alumno)

- **Estructura conceptual:** un modo CPA-seguro no debe permitir testar propiedades exactas del mensaje con una sola consulta.
- **Estructura conceptual:** en (a) aparece la relación $c_2 = r_2 \oplus r_1$; en (b) aparece $c_1 \oplus c_0 = F(k, m_1)$.
- **Por qué este método:** escogemos mensajes donde "bloques iguales" fuerzan un evento determinista ($c_2 = 0^\lambda$ o $c_2 = c_1 \oplus c_0$).
- **Por qué este método:** explotamos que $F(k, \cdot)$ es una permutación: entradas distintas $\Rightarrow$ salidas distintas.
- **Comprobaciones rápidas:** si $m_1 = m_2$ en (a), entonces $r_1 = r_2$ y $c_2 = 0^\lambda$.
- **Comprobaciones rápidas:** en (b), con $m_1 = 0^\lambda$ se cumple $c_1 \oplus c_0 = F(k, 0^\lambda)$ y $c_2 = F(k, m_2)$.
- **Errores comunes:** intentar "romper" con un evento que en el mundo aleatorio ocurra con probabilidad no despreciable.

- **Errores comunes:** olvidar que en CPA el adversario *elige* los mensajes, así que puede fijar m_1 convenientemente.
- **Conexiones globales:** muchos ataques CPA son de este tipo: construir un predicado que vale 1 en un mundo y 0 en el otro.
- **Ampliación (opcional):** estos modos fallan por falta de aleatorización/difusión por bloque: hay ecuaciones lineales visibles en el cifrado.

Ejercicio 4.

Demuestre formalmente que la seguridad pCCA implica la seguridad CCA.

Solución detallada

Interpretamos pCCA como IND$-CCA: la salida del oráculo de cifrado es indistinguible de una cadena uniforme (con un oráculo de descifrado consistente).

Sea Π un esquema. Para $b \in \{0,1\}$ definimos librerías CCA (*left-or-right*):

$$\mathcal{L}^{\Pi}_{\text{cca-}b}:\ \underline{\text{EAVESDROP}(m_0, m_1)}\ \text{return Enc}(k, m_b),$$

y `DECRYPT` devuelve $\text{Dec}(k,\cdot)$ salvo que se prohíbe descifrar el ciphertext de desafío. Definimos librerías pCCA (real vs ideal):

- $\mathcal{L}^{\Pi}_{\text{pcca-real}}$: `EAVESDROP`(m) devuelve $\text{Enc}(k,m)$ y `DECRYPT` devuelve $\text{Dec}(k,\cdot)$ (con la misma restricción sobre el ciphertext de desafío).
- $\mathcal{L}^{\Pi}_{\text{pcca-\$}}$: `EAVESDROP`$(m)$ elige $c \leftarrow \{0,1\}^{|c|}$ uniforme, almacena el par (c,m), devuelve c; `DECRYPT`(c) devuelve m si c está en la tabla y $\perp$ en otro caso (y también rechaza el ciphertext de desafío).

La seguridad pCCA dice que para todo distinguidor eficiente $\mathcal{D}$,

$$\left|\Pr\left[\mathcal{D} \diamond \mathcal{L}^{\Pi}_{\text{pcca-real}} \Rightarrow 1\right] - \Pr\left[\mathcal{D} \diamond \mathcal{L}^{\Pi}_{\text{pcca-\$}} \Rightarrow 1\right]\right|$$

es despreciable.

Ahora tomemos un distinguidor arbitrario $\mathcal{A}$ contra CCA (que hace consultas `EAVESDROP`(m_0,m_1) y `DECRYPT`). Construimos dos distinguidoras $\mathcal{D}_0, \mathcal{D}_1$ contra pCCA:

- $\mathcal{D}_0$ ejecuta $\mathcal{A}$ y responde a `EAVESDROP`(m_0,m_1) llamando a `EAVESDROP`(m_0) en la librería pCCA; reenvía `DECRYPT` tal cual.
- $\mathcal{D}_1$ igual, pero en `EAVESDROP`(m_0,m_1) llama a `EAVESDROP`(m_1).

Entonces:

- En el mundo **real** pCCA, $\mathcal{D}_0$ ve exactamente $\mathcal{L}^{\Pi}_{\text{cca-0}}$ y $\mathcal{D}_1$ ve exactamente $\mathcal{L}^{\Pi}_{\text{cca-1}}$, luego
$$\Pr\left[\mathcal{D}_0 \diamond \mathcal{L}^{\Pi}_{\text{pcca-real}} \Rightarrow 1\right] = \Pr\left[\mathcal{A} \diamond \mathcal{L}^{\Pi}_{\text{cca-0}} \Rightarrow 1\right],$$
y análogamente para $b = 1$.
- En el mundo **ideal** pCCA, la salida de `EAVESDROP`(m) es uniforme e independiente de m, por lo que $\mathcal{D}_0$ y $\mathcal{D}_1$ inducen la *misma* distribución de vista para $\mathcal{A}$. Así,
$$\Pr\left[\mathcal{D}_0 \diamond \mathcal{L}^{\Pi}_{\text{pcca-\$}} \Rightarrow 1\right] = \Pr\left[\mathcal{D}_1 \diamond \mathcal{L}^{\Pi}_{\text{pcca-\$}} \Rightarrow 1\right].$$

Aplicando la desigualdad triangular:

$$\begin{aligned}
&\left|\Pr\left[\mathcal{A} \diamond \mathcal{L}^{\Pi}_{\text{cca-0}} \Rightarrow 1\right] - \Pr\left[\mathcal{A} \diamond \mathcal{L}^{\Pi}_{\text{cca-1}} \Rightarrow 1\right]\right| \\
&= \left|\Pr\left[\mathcal{D}_0 \diamond \mathcal{L}^{\Pi}_{\text{pcca-real}} \Rightarrow 1\right] - \Pr\left[\mathcal{D}_1 \diamond \mathcal{L}^{\Pi}_{\text{pcca-real}} \Rightarrow 1\right]\right| \\
&\leq \left|\Pr\left[\mathcal{D}_0 \diamond \mathcal{L}^{\Pi}_{\text{pcca-real}} \Rightarrow 1\right] - \Pr\left[\mathcal{D}_0 \diamond \mathcal{L}^{\Pi}_{\text{pcca-\$}} \Rightarrow 1\right]\right| \\
&\quad + \left|\Pr\left[\mathcal{D}_1 \diamond \mathcal{L}^{\Pi}_{\text{pcca-\$}} \Rightarrow 1\right] - \Pr\left[\mathcal{D}_1 \diamond \mathcal{L}^{\Pi}_{\text{pcca-real}} \Rightarrow 1\right]\right|.
\end{aligned}$$

Cada término es despreciable por pCCA, luego el lado izquierdo es despreciable: Π es CCA-seguro.

Notas didácticas (para el alumno)

- **Estructura conceptual:** pCCA (IND$-CCA) es "más fuerte": el cifrado parece uniforme, no solo indistinguible entre dos mensajes.
- **Estructura conceptual:** la prueba usa *envoltorios* $\mathcal{D}_0, \mathcal{D}_1$ que alimentan a $\mathcal{A}$.
- **Por qué este método:** si algo es indistinguible de uniforme, entonces dos instancias (con mensajes distintos) son indistinguibles entre sí.
- **Por qué este método:** la igualdad en el mundo ideal es exacta porque el cifrado no depende del mensaje.
- **Comprobaciones rápidas:** identifica dónde se usa la desigualdad triangular (separar real vs ideal en dos pasos).
- **Comprobaciones rápidas:** verifica que $\mathcal{D}_0$ simula CCA-0 y $\mathcal{D}_1$ simula CCA-1 en el mundo real.
- **Errores comunes:** olvidar que la librería ideal pCCA debe ser *consistente* con `DECRYPT` (tabla).
- **Errores comunes:** mezclar "CCA" con "CCA sin restricción": siempre está prohibido descifrar el ciphertext de desafío.
- **Conexiones globales:** este patrón se repite en muchas implicaciones: "distinguible de A vs B" se acota por (A vs ideal) + (ideal vs B).
- **Ampliación (opcional):** el mismo esquema prueba IND$-CPA ⇒ IND-CPA.

Ejercicio 5.

Demuestra que los siguientes modos de cifrado de bloques no tienen seguridad CCA. Para cada uno, describe un distinguidor y demuestra que es capaz de distinguir entre las librerías correspondientes.

1. Modo OFB
2. Modo CBC
3. Modo CTR

Solución detallada

En los tres casos usaremos la misma idea: **maleabilidad** + oráculo de descifrado.

Fijamos dos mensajes de un bloque (λ bits) con $m_0 \neq m_1$ y definimos

$$\Delta := m_0 \oplus m_1 \neq 0^\lambda.$$

El distinguidor $\mathcal{A}$ hace una consulta de desafío y luego una consulta al oráculo `DECRYPT` sobre un ciphertext *modificado* (distinto del de desafío), para recuperar el bit b con certeza.

(a) OFB no es CCA-seguro. En OFB, para un bloque, el cifrado tiene la forma $c = c_0 \| c_1$ con

$$c_0 = \mathrm{IV}, \qquad c_1 = r \oplus m_b, \quad r = F_k(\mathrm{IV}).$$

Ataque:

1. $c_0 \| c_1 \leftarrow \mathrm{EAVESDROP}(m_0, m_1)$.
2. Forma $\tilde{c} := c_0 \| (c_1 \oplus \Delta)$ y consulta $\tilde{m} \leftarrow \mathrm{DECRYPT}(\tilde{c})$.
3. Devuelve `true` si $\tilde{m} = m_0$ (equiv., si $\tilde{m} \oplus \Delta = m_1$).

Como $m(\tilde{c}) = (c_1 \oplus \Delta) \oplus r = (m_b \oplus r \oplus \Delta) \oplus r = m_b \oplus \Delta$, tenemos:

$$\tilde{m} = \begin{cases} m_1, & b = 0, \\ m_0, & b = 1. \end{cases}$$

Luego $\mathcal{A}$ recupera b con certeza y su ventaja es 1.

(b) CBC no es CCA-seguro. Para un bloque en CBC:

$$c_0 = \mathrm{IV}, \qquad c_1 = F_k(m_b \oplus c_0), \qquad \mathrm{Dec} : m = F_k^{-1}(c_1) \oplus c_0.$$

Ataque:

1. $c_0 \| c_1 \leftarrow \mathrm{EAVESDROP}(m_0, m_1)$.
2. Forma $\tilde{c} := (c_0 \oplus \Delta) \| c_1$ y consulta $\tilde{m} \leftarrow \mathrm{DECRYPT}(\tilde{c})$.
3. Devuelve `true` si $\tilde{m} = m_0$.

Se cumple

$$\tilde{m} = F_k^{-1}(c_1) \oplus (c_0 \oplus \Delta) = (m_b) \oplus \Delta.$$

Así, igual que antes, si $b = 0$ entonces $\tilde{m} = m_1$ y si $b = 1$ entonces $\tilde{m} = m_0$, y la ventaja es 1.

(c) CTR no es CCA-seguro. En CTR, para un bloque:

$$c_0 = \mathrm{IV}, \qquad c_1 = F_k(\mathrm{IV} + 1) \oplus m_b, \qquad m = c_1 \oplus F_k(\mathrm{IV} + 1).$$

Ataque idéntico al de OFB:

1. $c_0 \| c_1 \leftarrow \mathrm{EAVESDROP}(m_0, m_1)$.
2. $\tilde{c} := c_0 \| (c_1 \oplus \Delta)$; consulta $\tilde{m} \leftarrow \mathrm{DECRYPT}(\tilde{c})$.
3. Decide b comparando $\tilde{m}$ con m_0, m_1.

Como en CTR la descifrado es XOR con el mismo flujo, también queda $\tilde{m} = m_b \oplus \Delta$ y la ventaja es 1.

Notas didácticas (para el alumno)

- **Estructura conceptual:** en CCA el atacante tiene `DECRYPT` (salvo el ciphertext de desafío), lo cual amplifica fallos de maleabilidad.
- **Estructura conceptual:** OFB/CTR son "flujo XOR"; CBC permite modificar el primer bloque modificando el IV.
- **Por qué este método:** elegimos $\Delta = m_0 \oplus m_1$ para que $m_b \oplus \Delta$ revele el otro mensaje.
- **Por qué este método:** modificamos el ciphertext de forma que siga siendo válido pero distinto del desafío (consulta permitida).
- **Comprobaciones rápidas:** en flujo XOR, cambiar c_1 a $c_1 \oplus \Delta$ cambia el claro a $m \oplus \Delta$.
- **Comprobaciones rápidas:** en CBC (un bloque), cambiar c_0 a $c_0 \oplus \Delta$ cambia el claro a $m \oplus \Delta$.
- **Errores comunes:** intentar descifrar directamente el ciphertext de desafío (está prohibido).
- **Errores comunes:** olvidar exigir $m_0 \neq m_1$ para que $\Delta \neq 0^\lambda$.
- **Conexiones globales:** por eso en la práctica se usa cifrado *autenticado* (AEAD): confidencialidad + integridad.
- **Ampliación (opcional):** estos ataques formalizan que estos modos son IND-CPA (bajo PRP) pero no IND-CCA sin autenticación.

Ejercicio 6.

Demuestra que el modo OFB es maleable.

Solución detallada

Recordemos que en OFB (para un bloque) el cifrado tiene la forma

$$c = c_0 \| c_1, \qquad c_0 = \mathrm{IV}, \qquad c_1 = r \oplus m, \quad r = F_k(\mathrm{IV}).$$

Sea un ciphertext válido $c_0 \| c_1$ que cifra un mensaje desconocido m. Sea $\Delta \in \{0,1\}^\lambda$ una máscara elegida por el atacante.
Definimos el ciphertext modificado

$$\tilde{c} := c_0 \| (c_1 \oplus \Delta).$$

Al descifrar,

$$\tilde{m} = (c_1 \oplus \Delta) \oplus r = (m \oplus r \oplus \Delta) \oplus r = m \oplus \Delta.$$

Es decir, **sin conocer** k **ni** m, el atacante puede transformar un cifrado de m en un cifrado de $m \oplus \Delta$ de manera controlada. Esto es exactamente maleabilidad: existe una transformación eficiente sobre ciphertexts que induce una relación predecible entre claros.

Notas didácticas (para el alumno)

- **Estructura conceptual:** "maleable" significa: puedo modificar un ciphertext para forzar una relación conocida entre claros.
- **Estructura conceptual:** OFB cifra como $c_1 = m \oplus r$ (flujo XOR), por eso hereda maleabilidad de XOR.
- **Por qué este método:** si haces $c_1 \mapsto c_1 \oplus \Delta$, el claro hace exactamente $m \mapsto m \oplus \Delta$.
- **Por qué este método:** la prueba es una cuenta de una línea: $(m \oplus r \oplus \Delta) \oplus r = m \oplus \Delta$.
- **Comprobaciones rápidas:** si Δ tiene un 1 en una posición, ese bit del claro se invierte.
- **Comprobaciones rápidas:** el IV no cambia el argumento: el flujo r se cancela en el XOR.
- **Errores comunes:** creer que "maleable" implica "no CPA-seguro": OFB puede ser CPA-seguro pero sigue siendo maleable.
- **Errores comunes:** confundir maleabilidad con "recuperar el mensaje": aquí no se aprende m, solo se fuerza una relación.
- **Conexiones globales:** maleabilidad + `DECRYPT` (CCA) $\Rightarrow$ ataques como los del ejercicio anterior.
- **Ampliación (opcional):** la solución práctica es añadir autenticación (MAC/AEAD) para detectar modificaciones.

Criptografía – Grado en Ingeniería de la Ciberseguridad

Hoja de problemas 4: MAC y propiedades de funciones hash

Escuela de Ingeniería Informática

Objetivos.

1. Practicar la definición de seguridad de **MAC** mediante **librerías intercambiables** (`GETTAG`/`CHECKTAG`) y construir distinguidoras.
2. Distinguir claramente entre propiedades de **PRF/PRP** y propiedades de **hash** (CR/-TR/PR).
3. Entrenar reducciones sencillas ("si rompes X, entonces rompes Y") y contraejemplos.

Ejercicio 1.

Considera el siguiente esquema MAC, donde F es una PRF segura con $\texttt{in} = \texttt{out} = \lambda$.

Gen:	$\mathrm{MAC}(k, m_1 \| m_2 \| \cdots \| m_\ell)$: // cada m_i tiene λ bits
$k \leftarrow \{0,1\}^\lambda$	$m^* := 0^\lambda$
return k	for $i := 1$ to ℓ:
	$\quad m^* := m^* \oplus m_i$
	return $F(k, m^*)$

Demuestra que el esquema no es un MAC seguro construyendo un programa que distinga las dos librerías relevantes de la definición.

Solución detallada

Usaremos el modelo estándar de librerías para MAC:

- $\mathcal{L}_{\text{mac-real}}$: $\texttt{GETTAG}(m)$ devuelve $t := \mathrm{MAC}(k, m)$ y $\texttt{CHECKTAG}(m, t)$ acepta si t es la etiqueta correcta para m.
- $\mathcal{L}_{\text{mac-fake}}$: $\texttt{GETTAG}(m)$ devuelve t y guarda (m, t) en una tabla T; $\texttt{CHECKTAG}(m, t)$ acepta *solo* si $(m, t) \in T$.

Observación clave: la etiqueta depende solo de la XOR de los bloques:

$$\mathrm{MAC}(k, m_1 \| \cdots \| m_\ell) = F\big(k,\, m_1 \oplus \cdots \oplus m_\ell\big).$$

Definimos el adversario $\mathcal{A}$:

$$\mathcal{A}: \quad t := \texttt{GETTAG}(0^\lambda); \quad \text{return } \texttt{CHECKTAG}(1^\lambda \| 1^\lambda,\, t).$$

Analizamos su comportamiento:

- En $\mathcal{L}_{\text{mac-real}}$, $\texttt{GETTAG}(0^\lambda)$ devuelve

$$t = \text{MAC}(k, 0^\lambda) = F(k, 0^\lambda).$$

Además,

$$\text{MAC}(k, 1^\lambda \| 1^\lambda) = F(k, 1^\lambda \oplus 1^\lambda) = F(k, 0^\lambda) = t,$$

así que `CHECKTAG` acepta con probabilidad 1. Por tanto,

$$\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{mac-real}} \Rightarrow \text{true}] = 1.$$

- En $\mathcal{L}_{\text{mac-fake}}$, tras $\texttt{GETTAG}(0^\lambda)$ la tabla contiene solo el par $(0^\lambda, t)$. Como $(1^\lambda \| 1^\lambda, t) \notin T$, `CHECKTAG` rechaza con probabilidad 1. Por tanto,

$$\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{mac-fake}} \Rightarrow \text{true}] = 0.$$

Luego $\mathcal{A}$ distingue ambas librerías con ventaja 1, y el esquema **no** es un MAC seguro.

Notas didácticas (para el alumno)

- **Estructura conceptual:** un MAC seguro impide forjar una etiqueta válida para un mensaje no consultado.
- **Estructura conceptual:** aquí la etiqueta solo depende de $m_1 \oplus \cdots \oplus m_\ell$.
- **Por qué este método:** buscamos dos mensajes distintos con la misma XOR para reutilizar una etiqueta.
- **Por qué este método:** 0^λ y $1^\lambda \| 1^\lambda$ tienen la misma XOR (0^λ).
- **Comprobaciones rápidas:** verifica $1^\lambda \oplus 1^\lambda = 0^\lambda$.
- **Comprobaciones rápidas:** en la librería `fake`, solo se aceptan pares exactamente consultados.
- **Errores comunes:** pensar que "usar una PRF" basta: el problema es la *compresión lineal* por XOR.
- **Errores comunes:** olvidar que el mensaje consultado y el mensaje forjado deben ser distintos.
- **Conexiones globales:** esto es un ejemplo de maleabilidad algebraica: si el resumen es lineal, hay forjas.
- **Ampliación (opcional):** compara con CBC-MAC: ahí se evita esta linealidad encadenando con F en cada bloque.

Ejercicio 2.

Considera el siguiente esquema MAC, donde F es una PRF segura con $\texttt{in} = \texttt{out} = \lambda$.

Gen:	$\underline{\text{MAC}(k, m_1 \| m_2 \| \cdots \| m_\ell)}$: // cada m_i tiene λ bits
$k \leftarrow \{0,1\}^\lambda$	$t := 0^\lambda$
return k	for $i := 1$ to ℓ:
	$t := t \oplus F(k, m_i)$
	return t

Demuestra que el esquema no es un MAC seguro construyendo un programa que distinga las dos librerías relevantes de la definición.

Solución detallada

De nuevo usamos $\mathcal{L}_{\text{mac-real}}$ y $\mathcal{L}_{\text{mac-fake}}$ como en el ejercicio anterior.

Observación clave: si el mensaje tiene dos bloques iguales, la etiqueta se anula:

$$\text{MAC}(k, m\|m) = F(k, m) \oplus F(k, m) = 0^\lambda.$$

Definimos el adversario $\mathcal{A}$ que no hace consultas a `GETTAG`:

$$\mathcal{A}: \quad \text{return } \texttt{CHECKTAG}(0^\lambda \| 0^\lambda,\, 0^\lambda).$$

Análisis:

- En $\mathcal{L}_{\text{mac-real}}$, como

  $$\text{MAC}(k, 0^\lambda \| 0^\lambda) = F(k, 0^\lambda) \oplus F(k, 0^\lambda) = 0^\lambda,$$

  la llamada `CHECKTAG` acepta con probabilidad 1. Por tanto,

  $$\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{mac-real}} \Rightarrow \text{true}] = 1.$$

- En $\mathcal{L}_{\text{mac-fake}}$, como $\mathcal{A}$ no ha llamado a `GETTAG`, la tabla T está vacía y necesariamente $(0^\lambda \| 0^\lambda, 0^\lambda) \notin T$. Por tanto,

  $$\Pr[\mathcal{A} \diamond \mathcal{L}_{\text{mac-fake}} \Rightarrow \text{true}] = 0.$$

Luego la ventaja es 1 y el esquema **no** es un MAC seguro.

Notas didácticas (para el alumno)

- **Estructura conceptual:** en un MAC, sin `GETTAG` no deberías poder hacer que `CHECKTAG` acepte.
- **Estructura conceptual:** aquí hay una cancelación algebraica: $F(k, m) \oplus F(k, m) = 0^\lambda$.
- **Por qué este método:** buscamos un mensaje cuya etiqueta sea *predecible* sin conocer k.
- **Por qué este método:** dos bloques iguales fuerzan etiqueta 0^λ.
- **Comprobaciones rápidas:** comprueba que el bucle empieza con $t = 0^\lambda$.
- **Comprobaciones rápidas:** en `fake`, si $T = \varnothing$ todo `CHECKTAG` debe fallar.

- **Errores comunes:** creer que "XOR de PRF" hereda seguridad MAC (no, depende de cómo se usa).
- **Errores comunes:** olvidar que el adversario puede elegir mensajes con estructura especial (como $m\|m$).
- **Conexiones globales:** esto ilustra por qué MACs reales suelen usar encadenamiento y longitudes.
- **Ampliación (opcional):** si permitieses longitud variable, también aparecen forjas por reordenación si no se fija el formato.

Ejercicio 3.

Supongamos que MAC es un algoritmo MAC seguro. Definimos $\text{MAC}'(k,m) = \text{MAC}(k,m)\|\text{MAC}(k,m)$. Demuestra que MAC' también es un algoritmo MAC seguro.

Nota: MAC' no puede ser una PRF segura. Esto muestra que la seguridad MAC es diferente de la seguridad PRF.

Solución detallada

Sea $\Pi = (\text{Gen}, \text{MAC}, \text{Vrfy})$ un MAC seguro y definamos $\Pi' = (\text{Gen}, \text{MAC}', \text{Vrfy}')$ por

$$\text{MAC}'(k,m) = t\|t \quad \text{donde } t := \text{MAC}(k,m),$$

y

$$\text{Vrfy}'(k,m,t_1\|t_2) = 1 \iff (t_1 = t_2) \ \wedge \ \text{Vrfy}(k,m,t_1) = 1.$$

Demostramos seguridad por reducción: si existe un forjador $\mathcal{A}'$ que rompe Π', construiremos un forjador $\mathcal{A}$ que rompe Π.

Construcción de $\mathcal{A}$. $\mathcal{A}$ tiene acceso a `GETTAG` y `CHECKTAG` del MAC original Π. Simula para $\mathcal{A}'$ la interfaz de Π' así:

- Para responder $\texttt{GETTAG}(m)$ de $\mathcal{A}'$, $\mathcal{A}$ pide $t := \texttt{GETTAG}(m)$ al oráculo real y devuelve $t\|t$.
- Cuando $\mathcal{A}'$ termina y devuelve una supuesta forja (m^*, τ^*), donde $\tau^* = t_1\|t_2$, $\mathcal{A}$:
  - si $t_1 \neq t_2$, devuelve fracaso (esa *no* es una forja válida para Π'),
  - si $t_1 = t_2$, entonces devuelve como forja para Π el par (m^*, t_1).

Si $\mathcal{A}'$ tiene éxito en Π', necesariamente $t_1 = t_2$ y $\text{Vrfy}(k, m^*, t_1) = 1$, con m^* no consultado antes. Por tanto, (m^*, t_1) es una forja válida para Π con la misma probabilidad. Esto contradice la seguridad de Π. Concluimos que Π' es un MAC seguro.

Notas didácticas (para el alumno)

- **Estructura conceptual:** "MAC seguro" significa que no se pueden forjar etiquetas nuevas, aunque veas muchas etiquetas válidas.
- **Estructura conceptual:** duplicar una etiqueta no añade información *utilizable* para forjar.

- **Por qué este método:** la forma estándar es reducción: forja de MAC' $\Rightarrow$ forja de MAC.
- **Por qué este método:** cualquier forja válida de MAC' debe tener dos mitades iguales.
- **Comprobaciones rápidas:** si $t_1 \neq t_2$, `Vrfy'` debe rechazar siempre.
- **Comprobaciones rápidas:** la simulación de `GETTAG` es exacta: devuelve $t\|t$.
- **Errores comunes:** confundir "parece aleatorio" (PRF) con "no se puede forjar" (MAC).
- **Errores comunes:** olvidar exigir que m^* no haya sido consultado en `GETTAG`.
- **Conexiones globales:** esta prueba es un ejemplo de "cerradura bajo codificación" para nociones de autenticidad.
- **Ampliación (opcional):** si en vez de duplicar hicieras $t\|0^\lambda$, la seguridad MAC seguiría (la parte extra es redundante).

Ejercicio 4.

Demuestre que un PRP básico no es resistente a colisiones. En otras palabras, si F es un PRP seguro, muestre cómo encontrar colisiones eficientemente en $H(x\|y) = F(x, y)$.

Solución detallada

Aquí $F(x, \cdot)$ es una permutación (PRP) para cada clave x. Como el adversario puede *elegir* x, puede invertir F siempre que conozca esa clave.

Algoritmo para encontrar una colisión:

1. Elige $x \in \{0,1\}^\lambda$ y $y \in \{0,1\}^\lambda$ arbitrarios y calcula
$$h := H(x\|y) = F(x, y).$$
2. Elige una clave distinta $x' \neq x$.
3. Calcula
$$y' := F^{-1}(x', h).$$

Entonces $(x\|y) \neq (x'\|y')$ pero

$$H(x\|y) = F(x, y) = h = F(x', y') = H(x'\|y').$$

Hemos construido una colisión en tiempo polinómico (una evaluación de F y una inversión de F).

Conclusión: aunque F sea un PRP seguro como cifrado, el hash $H(x\|y) = F(x, y)$ **no** es resistente a colisiones.

Notas didácticas (para el alumno)

- **Estructura conceptual:** "PRP seguro" no implica "hash CR"; son nociones distintas.
- **Estructura conceptual:** un PRP es invertible con la clave, y aquí la clave x está *dentro* de la entrada.
- **Por qué este método:** para una salida h, podemos forzar $F(x', y') = h$ eligiendo $y' = F^{-1}(x', h)$.
- **Por qué este método:** así garantizamos colisión exacta sin azar.
- **Comprobaciones rápidas:** revisa que $F(x', F^{-1}(x', h)) = h$ por definición de inversa.
- **Comprobaciones rápidas:** asegúrate de escoger $x' \neq x$ para que las entradas sean distintas.
- **Errores comunes:** pensar que "seguro" significa "seguro para cualquier uso" (no).
- **Errores comunes:** olvidar que en un hash la entrada es elegida por el atacante.
- **Conexiones globales:** por eso los hashes no se construyen metiendo "la clave" dentro del mensaje.
- **Ampliación (opcional):** si x fuese secreto (como clave externa), el argumento ya no aplica: sería un MAC/PRF, no un hash público.

Ejercicio 5.

Sean H_1 y H_2 dos funciones hash. Defina $H(x) = H_1(x) \| H_2(x)$.

- Demuestre que si al menos una de las funciones H_1 o H_2 es resistente a colisiones, entonces H es resistente a colisiones.
- Determine si se puede afirmar un resultado análogo en el caso de resistencia a TR y PR. Justifique su respuesta en cada caso.

Solución detallada

(i) Resistencia a colisiones (CR). Supongamos que existe una colisión $x \neq x'$ para H, es decir,

$$H(x) = H(x') \quad \Longleftrightarrow \quad H_1(x) \| H_2(x) = H_1(x') \| H_2(x').$$

Por igualdad de concatenaciones, esto implica simultáneamente

$$H_1(x) = H_1(x') \quad \text{y} \quad H_2(x) = H_2(x').$$

Luego $x \neq x'$ es una colisión para H_1 y también para H_2. Por contraposición: si al menos una de H_1 o H_2 es CR, entonces no puede existir una colisión para H y por tanto H es CR.

(ii) Resistencia a segundas imágenes (TR). Análogo: si para algún x existe $x' \neq x$ con $H(x) = H(x')$, entonces en particular $H_1(x) = H_1(x')$ y $H_2(x) = H_2(x')$. Por tanto, tal x' rompe TR de H_1 y de H_2. Luego, si al menos una de H_1 o H_2 es TR, entonces H también es TR.

(iii) Resistencia a preimagen (PR). Aquí el análisis *no* es análogo: que una de las componentes sea PR *no* fuerza que H sea PR. Damos un contraejemplo.
Sea $\hat{H}$ una función hash PR (por ejemplo, una función "ideal" para el argumento). Definimos sobre entradas $x = x_1 \| x_2$:

$$H_1(x_1\|x_2) := x_1\|\hat{H}(x_2), \qquad H_2(x_1\|x_2) := x_2.$$

Entonces H_1 hereda PR de $\hat{H}$ (invertir H_1 requiere invertir $\hat{H}$ en la segunda parte). Sin embargo,

$$H(x_1\|x_2) = H_1(x_1\|x_2)\|H_2(x_1\|x_2) = x_1\|\hat{H}(x_2)\|x_2.$$

Dado un valor $y = H(x)$ (proveniente de una entrada aleatoria), una preimagen se obtiene trivialmente leyendo x_1 y x_2 de y: si $y = y_1\|y_2\|y_3$ con $y_1 = x_1$ y $y_3 = x_2$, entonces $x' := y_1\|y_3$ cumple $H(x') = y$. Por tanto H **no** es PR, aunque H_1 sí lo sea.

Notas didácticas (para el alumno)

- **Estructura conceptual:** concatenar fuerza que cualquier colisión en H sea colisión en *ambas* componentes.
- **Estructura conceptual:** para TR ocurre lo mismo: una segunda imagen para H induce una segunda imagen para cada componente.
- **Por qué este método:** en CR/TR basta usar "igualdad de concatenaciones" $\Rightarrow$ igualdad componente a componente.
- **Por qué este método:** para PR buscamos un contraejemplo donde una parte del output *revele* la entrada.
- **Comprobaciones rápidas:** si $A\|B = A'\|B'$ (con longitudes fijas), entonces $A = A'$ y $B = B'$.
- **Comprobaciones rápidas:** en el contraejemplo, observa que $H(x)$ incluye x_2 explícitamente.
- **Errores comunes:** confundir "si una parte es PR" con "todo el vector es PR" (no necesariamente).
- **Errores comunes:** usar sin cuidado la cadena CR $\Rightarrow$ TR $\Rightarrow$ PR sin revisar el contexto del enunciado.
- **Conexiones globales:** la seguridad de hash suele requerir diseño que no filtre bits de la entrada.
- **Ampliación (opcional):** si ambas componentes son PR y "parecen independientes", entonces H suele ser PR, pero eso ya requiere hipótesis extra (no vale "una sola").

Ejercicio 6.

Sea H una función hash resistente a colisiones y defina $\hat{H}(x) = H(H(x))$. ¿Es $\hat{H}$ necesariamente resistente a colisiones?

Solución detallada

Sí. Probamos por reducción al absurdo.
Supongamos que existe una colisión para $\hat{H}$: existen $x \neq x'$ tales que

$$\hat{H}(x) = \hat{H}(x') \iff H(H(x)) = H(H(x')).$$

Sea $y := H(x)$ y $y' := H(x')$. Hay dos casos:

Caso 1: $y = y'$**.** Entonces $H(x) = H(x')$ con $x \neq x'$, es una colisión para H.

Caso 2: $y \neq y'$**.** La igualdad $H(y) = H(y')$ con $y \neq y'$ es una colisión para H.

En ambos casos obtenemos una colisión para H, contradiciendo que H sea resistente a colisiones. Por tanto, $\hat{H}$ es necesariamente resistente a colisiones.

Notas didácticas (para el alumno)

- **Estructura conceptual:** $\hat{H}$ aplica H dos veces; una colisión arriba debe "provenir" de una colisión de H en algún nivel.
- **Estructura conceptual:** el truco es definir $y = H(x)$ y $y' = H(x')$ y separar casos.
- **Por qué este método:** el caso $y = y'$ da colisión directa en H; el caso $y \neq y'$ da colisión en la segunda aplicación.
- **Por qué este método:** es una reducción estándar: "romper $\hat{H} \Rightarrow$ romper H".
- **Comprobaciones rápidas:** comprueba que $x \neq x'$ pero $H(x) = H(x')$ ya es una colisión válida.
- **Comprobaciones rápidas:** si $y \neq y'$ y $H(y) = H(y')$, eso también es colisión.
- **Errores comunes:** asumir sin justificar que $H(x) \neq H(x')$; hay que partir en dos casos.
- **Errores comunes:** confundir "colisión en $\hat{H}$" con "colisión en la primera capa" (puede ser en la segunda).
- **Conexiones globales:** construir nuevas funciones hash por composición suele preservar CR, pero no siempre preserva otras propiedades.
- **Ampliación (opcional):** pregunta interesante: ¿preserva PR o TR? Ahí el análisis cambia (no es tan directo).

Ejercicio 7.

Estudia la resistencia a segundas imágenes (TR) y a preimágenes (PR) de la función hash: dada $x = b_n b_{n-1} \cdots b_0$, sea z el entero que representa x (binario habitual), se calcula $r = 2z + 2^{25}$, se escribe en binario obteniendo w y se define $H(x)$ tomando los 8 bits más bajos de w.

Solución detallada

Escribimos z en base 2:

$$z = b_n 2^n + b_{n-1} 2^{n-1} + \cdots + b_1 2^1 + b_0 2^0.$$

Entonces

$$2z = b_n 2^{n+1} + b_{n-1}2^n + \cdots + b_1 2^2 + b_0 2^1.$$

Sumar 2^{25} no afecta a los 8 bits menos significativos (porque $25 > 7$), así que los 8 bits menos significativos de $r = 2z + 2^{25}$ son exactamente los de $2z$. Pero $2z$ es el binario de z desplazado una posición a la izquierda, por lo que sus 8 bits menos significativos son:

$$(b_6\, b_5\, b_4\, b_3\, b_2\, b_1\, b_0\, 0).$$

En particular,

$$H(b_n b_{n-1} \cdots b_0) = b_6 b_5 b_4 b_3 b_2 b_1 b_0 0.$$

No es resistente a preimagen (PR). Dado $y = y_7 y_6 \cdots y_0$ con $y_0 = 0$, podemos construir una preimagen en tiempo constante: elige, por ejemplo, una entrada de 9 bits

$$x' := 11 y_7 y_6 y_5 y_4 y_3 y_2 y_1,$$

es decir, fija $b_6 = y_7, \ldots, b_0 = y_1$ y el resto arbitrario. Entonces $H(x') = y$. (De hecho, si al atacante se le da $y = H(x)$ para un x aleatorio, siempre tendrá $y_0 = 0$ y el ataque funciona siempre.) Por tanto, existe un inversor eficiente y H no es PR.

No es resistente a segundas imágenes (TR). Fija una entrada $x = b_n \cdots b_7\, b_6 \cdots b_0$. Como H solo depende de $b_6, \ldots, b_0$, basta cambiar un bit alto (por ejemplo b_7) para obtener otra entrada con el mismo hash:

$$x' := b_n \cdots (b_7 \oplus 1)\, b_6 \cdots b_0.$$

Entonces $x' \neq x$ pero $H(x') = H(x)$. Por tanto, H no es TR.

Notas didácticas (para el alumno)

- **Estructura conceptual:** multiplicar por 2 en binario es desplazar a la izquierda y añadir un 0 al final.
- **Estructura conceptual:** tomar los 8 bits menos significativos significa que solo importa una ventana de bits de la entrada.
- **Por qué este método:** reescribimos $r = 2z + 2^{25}$ para ver qué bits afectan realmente a la salida.
- **Por qué este método:** 2^{25} no cambia los bits $0, \ldots, 7$, así que se ignora para H.
- **Comprobaciones rápidas:** confirma que el último bit del hash siempre es 0.
- **Comprobaciones rápidas:** verifica que H depende solo de $b_6, \ldots, b_0$ (7 bits) y un 0 fijo.
- **Errores comunes:** pensar que sumar 2^{25} "mezcla" bits bajos; no lo hace.
- **Errores comunes:** intentar invertir recuperando *todos* los bits: no hace falta, hay infinitas preimágenes.
- **Conexiones globales:** si un hash ignora muchos bits de la entrada, TR/PR se rompen fácilmente.
- **Ampliación (opcional):** incluso CR falla: basta fijar $b_6, \ldots, b_0$ y variar un bit alto para producir colisiones.

Criptografía – Grado en Ingeniería de la Ciberseguridad

Hoja de problemas 5: Establecimiento de clave, DDH/ElGamal y firmas RSA

Escuela de Ingeniería Informática

Objetivos.

1. Verificar **corrección** y **secreto** en protocolos sencillos de establecimiento de clave (y detectar fugas en el transcript).
2. Practicar reducciones basadas en **logaritmo discreto** y en la hipótesis **decisional de Diffie–Hellman (DDH)**.
3. Reconocer fallos estructurales de **RSA "de libro de texto"** en seguridad de firmas (forjas sin/ con una consulta).

Ejercicio 1.

Considere el siguiente esquema de establecimiento de clave:

- Alice escoge de manera uniforme $k, r \in \{0,1\}^n$ y envía $s = k \oplus r$ a Bob.
- Bob escoge de manera uniforme $t \in \{0,1\}^n$ y envía $u = s \oplus t$ a Alice.
- Alice computa $w = u \oplus r$ y envía w a Bob.
- La salida de Alice es k y la salida de Bob es $w \oplus t$.

Demuestra que Alice y Bob obtienen la misma clave. Analiza la seguridad de este protocolo frente a un adversario pasivo.

Solución detallada

Corrección. Sustituimos paso a paso:

$$s = k \oplus r, \qquad u = s \oplus t, \qquad w = u \oplus r.$$

La salida de Bob es

$$\begin{aligned} w \oplus t &= (u \oplus r) \oplus t \\ &= ((s \oplus t) \oplus r) \oplus t \\ &= (s \oplus r) \oplus (t \oplus t) \\ &= s \oplus r \\ &= (k \oplus r) \oplus r \\ &= k. \end{aligned}$$

Luego Alice y Bob obtienen la misma clave k.

Seguridad frente a adversario pasivo. Un adversario pasivo observa el transcript público: (s, u, w). A partir de estos tres valores puede recuperar la clave:

$$\begin{aligned} s \oplus u \oplus w &= s \oplus (s \oplus t) \oplus (u \oplus r) \\ &= (s \oplus s) \oplus t \oplus u \oplus r \\ &= t \oplus (s \oplus t) \oplus r \\ &= s \oplus r \\ &= k. \end{aligned}$$

Por tanto, el protocolo es **inseguro** frente a un adversario pasivo: el secreto k se filtra completamente en el transcript.

Notas didácticas (para el alumno)

- **Estructura conceptual:** primero se prueba corrección (misma clave) y luego secreto (nadie más aprende la clave).
- **Estructura conceptual:** XOR tiene cancelación: $x \oplus x = 0$ y $x \oplus 0 = x$.
- **Por qué este método:** se sustituye s, u, w en la salida de Bob y se simplifica con propiedades de XOR.
- **Por qué este método:** para seguridad pasiva basta ver si del transcript se puede aislar k.
- **Comprobaciones rápidas:** $w \oplus t = (u \oplus r) \oplus t$ y $u = s \oplus t$.
- **Comprobaciones rápidas:** prueba el ataque: $k = s \oplus u \oplus w$.
- **Errores comunes:** confundir "pasivo" (solo observa) con "activo" (modifica mensajes).
- **Errores comunes:** olvidar agrupar XORs y perder términos (mejor reordenar y cancelar).
- **Conexiones globales:** un KEX correcto no basta: el transcript no debe determinar la clave.
- **Ampliación (opcional):** Diffie–Hellman evita este tipo de cancelaciones porque sin DL/DDH no puedes "despejar" el secreto.

Ejercicio 2.

Supongamos que se te da un elemento X perteneciente al grupo cíclico generado por g, es decir, $X \in \langle g \rangle$. Tienes la capacidad de elegir cualquier otro elemento $X' \neq X$ del grupo y aprender el logaritmo discreto de X' con respecto a la base g. Demuestra cómo puedes utilizar esta capacidad para calcular el logaritmo discreto del elemento original X.

Solución detallada

Como $X \in \langle g \rangle$, existe un exponente desconocido x tal que $X = g^x$. Queremos calcular x.

Elige un y conocido con $y \not\equiv 0 \pmod{\text{ord}(g)}$ y define

$$X' := g^y \cdot X.$$

Podemos garantizar $X' \neq X$ eligiendo un y tal que $g^y \neq 1$.
Por hipótesis, podemos obtener $z = \log_g(X')$ tal que $X' = g^z$. Pero por construcción

$$X' = g^y \cdot g^x = g^{x+y},$$

luego

$$z \equiv x + y \pmod{\text{ord}(g)} \qquad \Rightarrow \qquad x \equiv z - y \pmod{\text{ord}(g)}.$$

Hemos calculado el logaritmo discreto de X usando una sola consulta al oráculo sobre un elemento distinto X'.

Notas didácticas (para el alumno)

- **Estructura conceptual:** DL: dado $X = g^x$, recuperar x (mod el orden del grupo).
- **Estructura conceptual:** multiplicar por g^y desplaza el exponente: $g^y \cdot g^x = g^{x+y}$.
- **Por qué este método:** fabricas X' con exponente relacionado con x para poder despejar.
- **Por qué este método:** el oráculo devuelve $z = \log_g(X')$ y luego haces $x = z - y$.
- **Comprobaciones rápidas:** confirma que $X' \in \langle g \rangle$ y que $X' \neq X$.
- **Comprobaciones rápidas:** revisa siempre el "mód ord(g)".
- **Errores comunes:** elegir $y \equiv 0$ y entonces $X' = X$ (prohibido).
- **Errores comunes:** olvidar que el logaritmo discreto está definido solo módulo el orden.
- **Conexiones globales:** este truco es típico en reducciones: re-randomizar instancias sin cambiar su "dificultad".
- **Ampliación (opcional):** también vale con $X' = X^a$ (si puedes elegir exponentes), porque $\log_g(X^a) = ax$.

Ejercicio 3.

Considera el siguiente esquema de cifrado asimétrico. La clave pública es (G, q, g, h) y la clave secreta x se generan como en El Gamal. Para cifrar un bit b:

- Si $b = 0$: elige $y \in \mathbb{Z}_q$ al azar y calcula $c_1 = g^y$, $c_2 = h^y$. Devuelve $c = (c_1, c_2)$.
- Si $b = 1$: elige $y, z \in \mathbb{Z}_q$ independientes y calcula $c_1 = g^y$, $c_2 = g^z$. Devuelve $c = (c_1, c_2)$.

(a) Escribir el algoritmo de descifrado y verificar que funciona.

(b) Mostrar que el esquema es CPA seguro bajo la hipótesis decisional de Diffie–Hellman.

Solución detallada

(a) Descifrado y corrección. En El Gamal, $h = g^x$ con $x \in \mathbb{Z}_q$ secreto. Dado $c = (c_1, c_2)$ definimos:

$$\mathrm{Dec}(x, (c_1, c_2)) := \begin{cases} 0, & \text{si } c_2 = c_1^x, \\ 1, & \text{si } c_2 \neq c_1^x. \end{cases}$$

Verificación:

- Si $b = 0$, entonces $c_1 = g^y$ y $c_2 = h^y = (g^x)^y = g^{xy}$. Además $c_1^x = (g^y)^x = g^{xy} = c_2$, así que devuelve 0 siempre.
- Si $b = 1$, entonces $c_1 = g^y$ y $c_2 = g^z$ con y, z independientes. La igualdad $c_2 = c_1^x$ equivale a $z \equiv xy \pmod{q}$, lo cual ocurre con probabilidad $1/q$.

Por tanto, el descifrado es correcto salvo con probabilidad $1/q$ (despreciable si q es grande).

(b) Seguridad CPA bajo DDH. Si existiera un adversario $\mathcal{A}$ que distingue cifrados de 0 y 1 con ventaja no despreciable, construiríamos un distinguidor $\mathcal{D}$ para DDH.

DDH. Dada (g, g^x, g^y, T), decidir si $T = g^{xy}$ (caso DDH-real) o $T = g^z$ uniforme (caso DDH-\$).

Construcción de $\mathcal{D}$. $\mathcal{D}$ recibe (g, A, B, T) donde $A = g^x$ y $B = g^y$. Define la clave pública para $\mathcal{A}$ como (G, q, g, h) con $h := A$. Entrega como ciphertext desafío

$$c^* := (c_1, c_2) := (B, T).$$

Si la instancia es DDH-real, entonces $T = g^{xy}$ y $c^* = (g^y, h^y)$, que es distribución de cifrado de $b = 0$. Si la instancia es DDH-\$, entonces $T = g^z$ independiente y $c^* = (g^y, g^z)$, que es distribución de cifrado de $b = 1$. Por tanto, $\mathcal{D}$ usa la salida de $\mathcal{A}$ para decidir DDH con la misma ventaja (hasta el término despreciable $1/q$ por la pequeña probabilidad de fallo de descifrado).

Notas didácticas (para el alumno)

- **Estructura conceptual:** $b = 0$ produce una tupla DH coherente (g^y, h^y); $b = 1$ rompe la coherencia.
- **Estructura conceptual:** el descifrado testea coherencia: $c_2 \stackrel{?}{=} c_1^x$.
- **Por qué este método:** DDH precisamente distingue g^{xy} de un g^z aleatorio.
- **Por qué este método:** el ciphertext desafío natural en la reducción es (B, T).
- **Comprobaciones rápidas:** en $b = 1$, la igualdad accidental ocurre con probabilidad $1/q$.
- **Comprobaciones rápidas:** en DDH-real, T actúa como h^y.
- **Errores comunes:** escribir “DDH-\$” sin escapar: debe ser “DDH-\$”.
- **Errores comunes:** mezclar corrección (aqui falla con $1/q$) con indistinguibilidad CPA.
- **Conexiones globales:** este es el patrón clásico: adversario CPA $\Rightarrow$ distinguidor DDH.

- **Ampliación (opcional):** se puede obtener corrección perfecta codificando el bit como pertenencia a un coset (variante de El Gamal).

Ejercicio 4.

Demuestra que la firma RSA "de libro de texto" no cumple con nuestra definición de seguridad para firmas digitales. Tu demostración solo debe utilizar la clave pública y no puede realizar llamadas a `GETSIG`.

Solución detallada

En RSA "de libro de texto", la clave pública es (N, e) y una firma sobre $m \in \mathbb{Z}_N$ es $\sigma := m^d$ mód N. La verificación acepta si $\sigma^e \equiv m \pmod{N}$.
Definimos el adversario $\mathcal{A}$:

$$\mathcal{A}: \ (N, e) := \texttt{GETVK}(); \ \sigma \leftarrow \mathbb{Z}_N; \ m := \sigma^e \text{ mód } N; \ \text{return } \texttt{VERSIG}(m, \sigma).$$

En la librería real, por construcción $m \equiv \sigma^e$ y `VERSIG` acepta siempre. En la librería ideal (sin llamadas a `GETSIG`), nunca se acepta porque no hay pares firmados. Luego $\mathcal{A}$ distingue con ventaja 1 y RSA "de libro" no es seguro bajo la definición.

Notas didácticas (para el alumno)

- **Estructura conceptual:** `VERSIG` en RSA "de libro" solo comprueba $\sigma^e \equiv m$.
- **Estructura conceptual:** si puedes elegir σ, puedes definir $m := \sigma^e$.
- **Por qué este método:** forja sin oráculo: solo usas `GETVK` y `VERSIG`.
- **Por qué este método:** no hay hash/padding que imponga estructura no manipulable.
- **Comprobaciones rápidas:** con $m = \sigma^e$, la verificación es tautológica.
- **Comprobaciones rápidas:** en la librería ideal, sin `GETSIG` no existe ningún par aceptable.
- **Errores comunes:** pensar que "RSA es difícil" implica seguridad de firma sin padding.
- **Errores comunes:** olvidar que el mensaje se modela como elemento de $\mathbb{Z}_N$ en esta versión.
- **Conexiones globales:** por eso se usa RSA-PSS o firmar un hash (FDH), no el mensaje directo.
- **Ampliación (opcional):** el ataque no usa factorizar N; explota solo la forma de verificación.

Ejercicio 5.

Demuestra que, en RSA "de libro de texto", es posible generar una firma válida para cualquier mensaje m (nunca antes firmado) realizando una única llamada a `GETSIG`. Sin embargo, dicha

llamada a `GETSIG` no debe ser directamente sobre m.

Solución detallada

Sea la clave pública (N, e) y la secreta d. `GETSIG` sobre M devuelve $\sigma := M^d \bmod N$. Sea $m \in \mathbb{Z}_N$ el mensaje que queremos firmar. Elegimos $r \in \mathbb{Z}_N^*$ y definimos

$$M := m \cdot (r^{-1})^e \bmod N.$$

Hacemos una sola consulta $\sigma := \texttt{GETSIG}(M) = M^d \bmod N$ y devolvemos

$$\sigma' := \sigma \cdot r \bmod N.$$

Comprobamos:

$$\begin{aligned}(\sigma')^e &\equiv (\sigma r)^e \equiv \sigma^e r^e \equiv M r^e \\ &\equiv \big(m(r^{-1})^e\big) r^e \equiv m \pmod N.\end{aligned}$$

Luego $\texttt{VERSIG}(m, \sigma')$ acepta: hemos firmado m con una consulta sobre $M \neq m$.

Notas didácticas (para el alumno)

- **Estructura conceptual:** RSA "de libro" es multiplicativo: $(ab)^d \equiv a^d b^d \pmod N$.
- **Estructura conceptual:** "enmascaras" m como $M = m(r^{-1})^e$ y luego "desenmascaras" la firma multiplicando por r.
- **Por qué este método:** quieres convertir una firma de oráculo en una firma de otro mensaje sin pedirla directamente.
- **Por qué este método:** el exponente e aparece en verificación, así que corriges con r^e.
- **Comprobaciones rápidas:** exige $r \in \mathbb{Z}_N^*$ para que exista r^{-1}.
- **Comprobaciones rápidas:** verifica $(\sigma r)^e \equiv \sigma^e r^e$.
- **Errores comunes:** escoger r no invertible (entonces no hay r^{-1}).
- **Errores comunes:** consultar `GETSIG` sobre m (prohibido).
- **Conexiones globales:** esta maleabilidad rompe EUF-CMA; el padding moderno la elimina.
- **Ampliación (opcional):** con RSA-PSS, la firma ya no es una potencia directa del mensaje y el ataque no funciona.

Criptografía – Grado en Ingeniería de la Ciberseguridad

Hoja de problemas 6: Compartición de secretos (Shamir), umbrales y seguridad

Escuela de Ingeniería Informática

Objetivos.

1. Diseñar y analizar esquemas de **compartición de secretos** (corrección, umbral y seguridad).
2. Practicar pruebas por **librerías intercambiables** y reducciones a primitivas (cifrado de un solo uso, DDH).
3. Identificar **fugas de información** en construcciones lineales y proponer distinguidoras.

1. Basado en el esquema de compartición de secretos de Shamir, diseña otro esquema tan seguro como el de Shamir, pero que sea capaz de compartir $r \geqslant 1$ secretos.

Solución detallada

La idea es *empaquetar* r secretos dentro de un único elemento del cuerpo finito y aplicar Shamir sin cambios.

Parámetros. Sea p primo (suficientemente grande) y trabajamos en $\mathbb{Z}_p$. Supongamos que los r secretos son cadenas $(m_1, \ldots, m_r)$, cada una de ℓ bits. Elegimos $p > 2^{r\ell}$ y fijamos una codificación inyectiva

$$\mathrm{Enc} : \{0,1\}^{r\ell} \to \mathbb{Z}_p, \qquad \mathrm{Dec} : \mathrm{Enc}(\{0,1\}^{r\ell}) \to \{0,1\}^{r\ell}$$

(típicamente, interpretar $m_1 \| \cdots \| m_r$ como entero en binario).

Esquema (umbral t de n).

- Share$(m_1, \ldots, m_r)$:
  I. Define $a_0 := \mathrm{Enc}(m_1 \| \cdots \| m_r) \in \mathbb{Z}_p$.
  II. Elige $a_1, \ldots, a_{t-1} \leftarrow \mathbb{Z}_p$ uniformes e independientes.
  III. Define el polinomio $f(x) = a_0 + a_1 x + \cdots + a_{t-1} x^{t-1} \in \mathbb{Z}_p[x]$.
  IV. Para cada $i = 1, \ldots, n$, devuelve la participación $s_i := (i, f(i))$.
- Reconstruct$(\{(i, f(i))\}_{i \in I})$: con $|I| \geq t$, interpola el único polinomio f de grado $\leq t-1$ que coincide en esos puntos y calcula

$$a_0 = f(0), \qquad (m_1, \ldots, m_r) = \mathrm{Dec}(a_0).$$

Corrección. Con t participaciones, la interpolación recupera f de manera única (polinomio de grado $\leq t-1$ con t puntos distintos), luego recupera $a_0 = f(0)$ y por tanto $(m_1, \ldots, m_r)$.

Seguridad. Sea U un conjunto de índices con $|U| < t$ (adversario pasivo ve menos de t participaciones). En Shamir, para cualquier valor del secreto a_0, el vector de participaciones $\{(i, f(i))\}_{i \in U}$ tiene la misma distribución (uniforme sobre un afín) cuando $a_1, \ldots, a_{t-1}$ son uniformes. Como nuestro esquema es literalmente Shamir aplicado al secreto $a_0 = \mathrm{Enc}(m_1 \| \cdots \| m_r)$, la vista de cualquier adversario con menos de t participaciones es **independiente** de $(m_1, \ldots, m_r)$.

Notas didácticas (para el alumno)

- **Estructura conceptual:** Shamir comparte un secreto como $f(0)$ y reparte puntos $f(i)$ de un polinomio aleatorio.
- **Estructura conceptual:** "compartir r secretos" puede hacerse codificando r secretos en un solo elemento a_0.
- **Por qué este método:** reutiliza la prueba de Shamir sin tocarla: solo cambia qué significa a_0.
- **Por qué este método:** la aleatoriedad sigue estando en $a_1, \ldots, a_{t-1}$.
- **Comprobaciones rápidas:** exige $p > 2^{r\ell}$ para que Enc sea inyectiva.
- **Comprobaciones rápidas:** con t puntos distintos, la interpolación determina f de forma única.
- **Errores comunes:** elegir p pequeño y perder invertibilidad de la codificación.
- **Errores comunes:** olvidar que los $x = i$ deben ser distintos en $\mathbb{Z}_p$ (toma $p > n$).
- **Conexiones globales:** es el mismo principio que "empaquetar" mensajes en un bloque antes de cifrar.
- **Ampliación (opcional):** existe "packed Shamir" donde los r secretos se ponen en varios valores $f(0), f(1), \ldots$ (más eficiente, pero requiere más cuidado).

2. Demuestra que si Σ es un esquema de cifrado con intercambiabilidad de un solo uso, entonces el siguiente esquema de compartición de secretos $\mathcal{S}$ con umbral 2 de 2 es seguro:

$\mathcal{M} = \Sigma.\mathcal{M}$	Share(m):	Reconstruct(s_1, s_2):
$t = 2$	$s_1 \leftarrow \Sigma.\mathsf{Gen}$	return $\Sigma.\mathsf{Dec}(s_1, s_2)$
$n = 2$	$s_2 \leftarrow \Sigma.\mathsf{Enc}(s_1, m)$	
	return (s_1, s_2)	

Solución detallada

Corrección. Por definición, $s_2 = \Sigma.\mathsf{Enc}(s_1, m)$, luego

$$\Sigma.\mathsf{Dec}(s_1, s_2) = \Sigma.\mathsf{Dec}(s_1, \Sigma.\mathsf{Enc}(s_1, m)) = m$$

por corrección del cifrado Σ.

Seguridad (umbral 2 de 2). Debemos ver que cualquier adversario que vea *una sola* participación (esto es, $|U| < 2$) no obtiene información sobre m. Los únicos casos son $U = \{1\}$ o $U = \{2\}$.

Caso $U = \{1\}$. La vista del adversario es $s_1 \leftarrow \Sigma.\mathsf{Gen}$, que no depende de m. Luego la distribución es idéntica para cualquier m_0, m_1.

Caso $U = \{2\}$. La vista del adversario es $s_2 = \Sigma.\mathsf{Enc}(s_1, m)$ donde $s_1 \leftarrow \Sigma.\mathsf{Gen}$. Si existiera un distinguidor $\mathcal{A}$ que, viendo solo s_2, distinguiese si se cifró m_0 o m_1 con ventaja no despreciable, construiríamos un distinguidor $\mathcal{D}$ que rompe la intercambiabilidad de un solo uso de Σ:

- $\mathcal{D}$ envía (m_0, m_1) al desafío de Σ y recibe un ciphertext c que es $\Sigma.\mathsf{Enc}(k, m_b)$ para un bit b oculto.
- $\mathcal{D}$ entrega c a $\mathcal{A}$ como si fuera s_2 y devuelve el bit que produce $\mathcal{A}$.

Entonces $\mathcal{D}$ distingue la librería de cifrado para m_0 y m_1 con la misma ventaja, contradiciendo la seguridad de Σ. Concluimos que $\mathcal{S}$ es seguro.

Notas didácticas (para el alumno)

- **Estructura conceptual:** umbral 2 de 2 significa: con 1 participación no aprendes nada; con 2 reconstruyes.
- **Estructura conceptual:** aquí s_1 es una clave y s_2 es el cifrado del secreto.
- **Por qué este método:** se hace por casos ($U = \{1\}$ y $U = \{2\}$) porque solo hay dos vistas posibles.
- **Por qué este método:** para $U = \{2\}$ la propiedad exacta necesaria es seguridad de cifrado de un solo uso.
- **Comprobaciones rápidas:** $U = \{1\}$: la clave s_1 no depende de m.
- **Comprobaciones rápidas:** $U = \{2\}$: lo único que ves es un ciphertext.
- **Errores comunes:** pensar que conocer s_2 y no conocer s_1 "no cuenta" como ataque (sí cuenta).
- **Errores comunes:** olvidar construir la reducción (cómo convertir un ataque a $\mathcal{S}$ en uno a Σ).
- **Conexiones globales:** muchos esquemas de secret sharing se construyen "envolviendo" el secreto con cifrado.
- **Ampliación (opcional):** este esquema es el análogo de "cifrado + entregar la clave por otro canal".

3. Diseña un esquema de compartición de secretos seguro con umbral 3 de 3, utilizando como base el esquema presentado en el problema anterior. Justifica rigurosamente tanto la corrección como la seguridad de tu construcción.

Solución detallada

Tomamos dos esquemas de cifrado simétrico Σ_1 y Σ_2 tales que:

- Σ_1 tiene **cifrados uniformes**: para cualquier clave k y cualquier mensaje x, el valor $\Sigma_1.\mathsf{Enc}(k, x)$ es uniforme en su espacio de ciphertexts (en particular, no depende de x).

- Σ_2 tiene **intercambiabilidad de un solo uso**.

Construcción $\mathcal{S}_{3,3}$. Para un secreto $m \in \Sigma_2.\mathcal{M}$:

- Share(m):
  I. $s_1 \leftarrow \Sigma_1.\mathsf{Gen}$.
  II. $k \leftarrow \Sigma_2.\mathsf{Gen}$.
  III. $s_2 \leftarrow \Sigma_1.\mathsf{Enc}(s_1, k)$.
  IV. $s_3 \leftarrow \Sigma_2.\mathsf{Enc}(k, m)$.
  V. return (s_1, s_2, s_3).
- Reconstruct(s_1, s_2, s_3):

$$k := \Sigma_1.\mathsf{Dec}(s_1, s_2), \qquad \text{return } \Sigma_2.\mathsf{Dec}(k, s_3).$$

Corrección. Si $s_2 = \Sigma_1.\mathsf{Enc}(s_1, k)$, entonces $\Sigma_1.\mathsf{Dec}(s_1, s_2) = k$. Si además $s_3 = \Sigma_2.\mathsf{Enc}(k, m)$, entonces $\Sigma_2.\mathsf{Dec}(k, s_3) = m$. Luego Reconstruct devuelve m.

Seguridad (cualquier subconjunto de tamaño < 3). Sea $U \subseteq \{1, 2, 3\}$ con $|U| \leq 2$. Consideramos los tres casos esenciales:

Caso $U = \{1, 2\}$. La vista es (s_1, s_2) donde $s_2 = \Sigma_1.\mathsf{Enc}(s_1, k)$. Por "cifrados uniformes", s_2 es uniforme e independiente de k y de m, incluso dado s_1. Luego (s_1, s_2) no revela información sobre m.

Caso $U = \{1, 3\}$. La vista es (s_1, s_3) donde s_1 es independiente y $s_3 = \Sigma_2.\mathsf{Enc}(k, m)$ con k secreto aleatorio. Si hubiera un distinguidor que, viendo (s_1, s_3), distinguiese entre m_0 y m_1, ignorando s_1 obtendríamos un distinguidor que rompe la intercambiabilidad de un solo uso de Σ_2 (es exactamente el caso "ver solo el ciphertext").

Caso $U = \{2, 3\}$. La vista es (s_2, s_3). Usamos un argumento híbrido:

I. Por "cifrados uniformes", podemos reemplazar $s_2 = \Sigma_1.\mathsf{Enc}(s_1, k)$ por un ciphertext uniforme $C \leftarrow \Sigma_1.\mathcal{C}$ sin cambiar la distribución.

II. Tras ese reemplazo, la vista es $(C, \Sigma_2.\mathsf{Enc}(k, m))$ donde C es independiente de todo. Entonces distinguir m_0 de m_1 equivale a distinguir $\Sigma_2.\mathsf{Enc}(k, m_0)$ de $\Sigma_2.\mathsf{Enc}(k, m_1)$, lo cual contradice la intercambiabilidad de un solo uso de Σ_2.

Concluimos que para cualquier $|U| \leq 2$ la vista es indistinguible entre secretos, y por tanto el esquema es seguro con umbral 3 de 3.

Notas didácticas (para el alumno)

- **Estructura conceptual:** para umbral 3 de 3, cualquier 1 o 2 participaciones deben ocultar el secreto.
- **Estructura conceptual:** dividimos el problema: una capa "esconde una clave" y otra capa cifra el mensaje.
- **Por qué este método:** la prueba va por casos ($\{1, 2\}$, $\{1, 3\}$, $\{2, 3\}$) porque son todas las vistas con 2 shares.

- **Por qué este método:** en $\{2,3\}$ usamos un híbrido: primero hacemos s_2 uniforme y luego aplicamos seguridad de Σ_2.
- **Comprobaciones rápidas:** Reconstruct hace dos descifrados en cadena: primero recupera k, luego recupera m.
- **Comprobaciones rápidas:** en $\{1,2\}$, s_2 debe ser independiente de k (ahí entra "cifrados uniformes").
- **Errores comunes:** olvidar justificar por qué puedes cambiar s_2 por un uniforme (esa es la hipótesis fuerte de Σ_1).
- **Errores comunes:** en $\{1,3\}$, pensar que ver un ciphertext es seguro "solo si conoces algo de la clave" (al revés).
- **Conexiones globales:** esto es "envolver" un secreto con cifrado y repartir la clave cifrada por separado.
- **Ampliación (opcional):** con más participantes, el mismo patrón se generaliza encadenando claves o usando Shamir para la clave.

4. Considera el siguiente esquema de compartición de secretos con umbral 3 de 3:

   Dado un grupo cíclico $G = \langle g \rangle$ de orden q primo,

$\mathcal{M} = \{0,1\}$	$\underline{\text{Share}(m)}$:	$\underline{\text{Reconstruct}(a,B,C)}$:
	$a, b, c \leftarrow \mathbb{Z}_q,\ c \neq ab$	return $B^a \stackrel{?}{=} C$
	If $m = 0$ return (a, g^b, g^c)	
	If $m = 1$ return (a, g^b, g^{ab})	

   Determina si el esquema es seguro. Si es seguro, proporciona una justificación formal. Si no es seguro, describe un distinguidor eficiente y justifica su capacidad de distinguir entre las librerías relevantes.

Solución detallada

Corrección. Si $m = 1$, entonces $B = g^b$ y $C = g^{ab}$, luego $B^a = (g^b)^a = g^{ab} = C$ y Reconstruct devuelve `true`. Si $m = 0$, entonces $C = g^c$ con $c \neq ab$, por tanto $g^c \neq g^{ab}$ (porque G tiene orden primo q) y Reconstruct devuelve `false`. Luego el esquema reconstruye correctamente el bit.

Seguridad (vistas de tamaño < 3). Hay tres tipos de vistas con a lo sumo 2 participaciones:

$$\{1,2\} :\ (a, g^b), \qquad \{1,3\} :\ (a, C), \qquad \{2,3\} :\ (g^b, C).$$

$\underline{\text{Caso } U = \{1,2\}.}$ La distribución de (a, g^b) es idéntica para $m = 0$ y $m = 1$ (no depende de m).

$\underline{\text{Caso } U = \{1,3\}.}$ Para $m = 1$, $C = g^{ab}$ con b uniforme. Para $m = 0$, $C = g^c$ con c uniforme sujeto a $c \neq ab$. En ambos casos, condicionado a un a fijo, C es "casi uniforme" en G:

- si $m = 1$ y b es uniforme en $\mathbb{Z}_q$, entonces ab es uniforme en $\mathbb{Z}_q$ cuando $a \neq 0$, y C es uniforme en G;

- si $m = 0$, entonces c es uniforme en $\mathbb{Z}_q \setminus \{ab\}$ y C es uniforme en $G \setminus \{g^{ab}\}$.

La distancia estadística entre "uniforme en G" y "uniforme en G salvo un elemento" es a lo sumo $1/q$. Además, el evento $a = 0$ ocurre con probabilidad $1/q$; en ese evento $m = 1$ fuerza $C = 1$ y $m = 0$ fuerza $C \neq 1$, lo cual introduce como mucho otro término $1/q$. Por tanto, cualquier distinguidor tiene ventaja acotada por $2/q$ en este caso.

Caso $U = \{2, 3\}$. La vista es (g^b, C). Si $m = 1$, es (g^b, g^{ab}); si $m = 0$, es (g^b, g^c) con $c \neq ab$. De nuevo, la única diferencia estadística viene de excluir un solo valor (y de la posibilidad $b = 0$), así que la ventaja de cualquier distinguidor está acotada por un término del orden de $1/q$. En particular, si q es exponencial en el parámetro de seguridad, esta ventaja es despreciable.

Conclusión. El esquema es **seguro** en el sentido habitual de indistinguibilidad (ventaja despreciable), con fuga estadística a lo sumo del orden de $1/q$ debida a la condición $c \neq ab$ y a los casos raros $a = 0$ o $b = 0$.

Notas didácticas (para el alumno)

- **Estructura conceptual:** el bit se codifica como "hay relación DH" ($m = 1$) o "no la hay" ($m = 0$).
- **Estructura conceptual:** Reconstruct solo comprueba si C coincide con B^a.
- **Por qué este método:** para umbral 3 de 3 basta analizar todas las vistas de 2 participantes.
- **Por qué este método:** la seguridad se reduce a que, sin los 3 datos, C se ve (casi) uniforme.
- **Comprobaciones rápidas:** $m = 1$: $B^a = (g^b)^a = g^{ab} = C$.
- **Comprobaciones rápidas:** $m = 0$: como $c \neq ab$, necesariamente $g^c \neq g^{ab}$.
- **Errores comunes:** olvidar los casos raros $a = 0$ o $b = 0$ (aparecen con probabilidad $1/q$).
- **Errores comunes:** confundir "perfectamente seguro" con "seguro con ventaja despreciable".
- **Conexiones globales:** esta construcción está muy cerca del patrón DDH: "g^{ab}" vs "g^c".
- **Ampliación (opcional):** si se muestrea $a, b \leftarrow \mathbb{Z}_q^*$ y $c \leftarrow \mathbb{Z}_q$ libre, se eliminan algunas fugas estadísticas.

5. Considera el siguiente esquema de compartición de secretos: para un mensaje $m \in \{0,1\}^{500}$,

Share: Con input $m = m_1 \| m_2 \| m_3 \| m_4 \| m_5$ con cada $|m_i| = 100$, devuelve como salida

$$s = (m_1,\ m_2 \oplus m_1,\ m_3 \oplus m_2,\ m_4 \oplus m_3,\ m_5 \oplus m_4).$$

(a) Determina el umbral del esquema y describe el algoritmo de recuperación del mensaje.

(b) Determina si el esquema es seguro. Si es seguro, proporciona una justificación formal. Si no es seguro, describe un distinguidor eficiente y justifica su capacidad de distinguir entre las librerías relevantes.

Solución detallada

(a) Umbral y reconstrucción. Denotemos las participaciones como

$$s_1 = m_1, \quad s_2 = m_2 \oplus m_1, \quad s_3 = m_3 \oplus m_2, \quad s_4 = m_4 \oplus m_3, \quad s_5 = m_5 \oplus m_4.$$

A partir de *todas* las participaciones se reconstruye en cadena:

$$\begin{aligned} m_1 &= s_1, \\ m_2 &= s_2 \oplus m_1, \\ m_3 &= s_3 \oplus m_2, \\ m_4 &= s_4 \oplus m_3, \\ m_5 &= s_5 \oplus m_4, \end{aligned}$$

y se devuelve $m = m_1 \| m_2 \| m_3 \| m_4 \| m_5$.
Observación: sin s_1 no puedes iniciar la cadena, y sin s_5 no puedes recuperar m_5, etc. En particular, faltando cualquier s_i queda algún bloque irrecuperable. Luego el umbral es 5 de 5.

(b) Seguridad. El esquema **no** es seguro, porque incluso una sola participación revela información del secreto. En concreto, $s_1 = m_1$ expone los primeros 100 bits del mensaje.

Distinguidor (ventaja 1) en la definición por librerías: elige

$$m^{(0)} := 0^{500}, \qquad m^{(1)} := 1^{100} \| 0^{400}.$$

Pide al desafío una compartición del mensaje $m^{(b)}$ y observa solo la primera participación ($U = \{1\}$). Recibe s_1 y decide:

$$\text{si } s_1 = 0^{100} \text{ devuelve } 0, \quad \text{si no, devuelve } 1.$$

Bajo $b = 0$ siempre $s_1 = 0^{100}$; bajo $b = 1$ siempre $s_1 = 1^{100}$. Por tanto distingue con probabilidad 1 y el esquema es inseguro.

Notas didácticas (para el alumno)

- **Estructura conceptual:** las participaciones son combinaciones lineales (XOR) de bloques consecutivos.
- **Estructura conceptual:** reconstrucción es una "cadena": cada bloque recupera el siguiente.
- **Por qué este método:** escribir s_i explícitamente en función de los m_i muestra qué se puede despejar.
- **Por qué este método:** para atacar seguridad basta ver si alguna share depende directamente del secreto.
- **Comprobaciones rápidas:** $s_1 = m_1$ **ya** revela 100 bits: esto mata la seguridad.
- **Comprobaciones rápidas:** sin s_1 no puedes recuperar m_2 porque $m_2 = s_2 \oplus m_1$.
- **Errores comunes:** confundir "umbral alto" (5 de 5) con "seguridad" (no implica ocultar con menos shares).

- **Errores comunes:** pensar que XOR "aleatoriza" por sí solo; si no hay aleatoriedad extra, no oculta.
- **Conexiones globales:** un esquema seguro necesita aleatoriedad (como coeficientes aleatorios en Shamir).
- **Ampliación (opcional):** si a cada ecuación le añades una máscara aleatoria compartida aparte, puedes acercarte a seguridad, pero entonces cambian umbral y reconstrucción.

Criptografía – Grado en Ingeniería de la Ciberseguridad

Hoja de problemas 7: Compromisos: hash, PRG y RSA; reducciones

Escuela de Ingeniería Informática

Objetivos.

1. Escribir formalmente esquemas de **compromiso** a partir de descripciones informales (Gen/Commit/Open/Verify).
2. Identificar qué propiedad criptográfica es necesaria para **hiding** y para **binding** (hash y PRG).
3. Practicar una **reducción** completa: romper binding ⇒ extraer una raíz RSA (descifrar sin clave privada).

1. Considera la descripción informal del siguiente esquema de compromiso:

   - Alice se compromete a un bit b generando dos cadenas de bits aleatorias R_1 y R_2.
   - Alice utiliza una función hash H para calcular $H(R_1\|R_2\|b)$.
   - Alice revela a Bob el valor del hash y la cadena R_1.
   - Cuando llega el momento de revelar el bit b, Alice muestra a Bob el mensaje $R_1\|R_2\|b$.
   - Bob comprueba que R_1 es el mismo de antes y que el valor del hash es el correcto.

   Escribe los algoritmos correspondientes al esquema descrito. ¿Cuál es el nivel mínimo de seguridad que debe tener la función hash (resistencia a colisiones, resistencia a segunda preimagen, resistencia a preimagen) para que este esquema cumpla las condiciones de hiding y binding? ¿Es necesaria la cadena R_2?

   Solución detallada

   Fijamos un parámetro de seguridad n y una función hash

   $$H : \{0,1\}^{2n+1} \to \{0,1\}^n.$$

   Interpretamos $R_1, R_2 \in \{0,1\}^n$ y $b \in \{0,1\}$.

   Algoritmos. Un esquema de compromiso se da por $\mathrm{Gen}, \mathrm{Commit}, \mathrm{Open}, \mathrm{Verify}$. Aquí no hay clave secreta, así que podemos tomar Gen trivial.

   - $\underline{\mathrm{Gen}(1^n)}$: return $\mathrm{ck} := \varepsilon$.
   - $\underline{\mathrm{Commit}(\mathrm{ck}, b)}$:

     $$R_1 \leftarrow \{0,1\}^n, \qquad R_2 \leftarrow \{0,1\}^n, \qquad h := H(R_1\|R_2\|b).$$

     return compromiso $\mathrm{com} := (h, R_1)$ y apertura $\mathrm{dec} := (R_2, b)$.

- Open(com, dec): return el par (R_2, b).
- Verify(com, dec): con com $= (h, R_1)$ y dec $= (R_2, b)$, acepta si y solo si

$$h = H(R_1 \| R_2 \| b).$$

Hiding: qué pide a H. El verificador ve (h, R_1) antes de la apertura. Para aprender b tendría que encontrar una apertura válida (R_2, b) tal que

$$H(R_1 \| R_2 \| b) = h.$$

Un ataque directo es hallar una preimagen compatible con el prefijo R_1. Por tanto, el requisito mínimo natural para hiding es **resistencia a preimagen**.

Binding: qué pide a H. Para romper binding, Alice debe abrir el mismo compromiso de dos formas distintas, por ejemplo

$$H(R_1 \| R_2 \| 0) = H(R_1 \| R_2' \| 1) = h,$$

con $(R_2, 0) \neq (R_2', 1)$. Esto es una **colisión**. Luego, para binding el requisito mínimo es **resistencia a colisiones**.

¿Es necesaria R_2? Sí. Si eliminamos R_2 y definimos $h := H(R_1 \| b)$, entonces como R_1 se revela, Bob puede computar

$$H(R_1 \| 0) \quad \text{y} \quad H(R_1 \| 1)$$

y comparar con h, recuperando b inmediatamente. Luego R_2 es necesaria para hiding.

Notas didácticas (para el alumno)

- **Estructura conceptual:** compromiso = hiding (no se ve b) + binding (no puedes cambiar b).
- **Estructura conceptual:** aquí com $= (h, R_1)$ y la apertura real es (R_2, b).
- **Por qué este método:** escribir Gen/Commit/Verify evita malentendidos en la definición.
- **Comprobaciones rápidas:** si quitas R_2, Bob prueba $b = 0, 1$ y rompe hiding.
- **Comprobaciones rápidas:** dos aperturas distintas para el mismo h implican colisión.
- **Errores comunes:** confundir preimagen con colisión (una no implica la otra).
- **Errores comunes:** olvidar que R_1 es público y fija el prefijo del hash.
- **Conexiones globales:** este patrón es "hash-and-open"; el hash funciona como huella.
- **Ampliación (opcional):** en oráculo aleatorio, con R_2 largo el hiding es casi perfecto.

2. Considera la descripción informal del siguiente esquema de compromiso:

- Se establece como clave pública una cadena de bits aleatoria R de longitud $3n$.
- Alice escoge una cadena de bits x aleatoria de longitud n y calcula $G(x)$, donde $G : \{0,1\}^n \to \{0,1\}^{3n}$ es un generador pseudoaleatorio seguro.
- Alice se compromete a un bit b. Si $b = 1$, entonces envía $G(x)$ a Bob. Si $b = 0$, envía $G(x) \oplus R$.
- Para revelar el bit al que se ha comprometido, Alice revela x y Bob verifica si ha recibido $G(x)$ o $G(x) \oplus R$.

Escribe los algoritmos correspondientes al esquema descrito. Determina si este esquema cumple las propiedades de hiding y binding.

Solución detallada

Algoritmos.

- Gen(1^n): elige $R \leftarrow \{0,1\}^{3n}$ y return ck := R.
- Commit(R, b): elige $x \leftarrow \{0,1\}^n$ y calcula $y := G(x)$. Define

$$c := \begin{cases} y, & \text{si } b = 1, \\ y \oplus R, & \text{si } b = 0, \end{cases}$$

  y return com := c y apertura dec := (b, x).
- Verify($R, c, (b, x)$): acepta si

$$(b = 1 \ \wedge \ c = G(x)) \ \text{ o } \ (b = 0 \ \wedge \ c = G(x) \oplus R).$$

Hiding. Usamos un híbrido: reemplaza $G(x)$ por $U \leftarrow \{0,1\}^{3n}$ uniforme. Como G es PRG seguro, el adversario no distingue el mundo real del ideal con ventaja no despreciable. En el mundo ideal, si $b = 1$ entonces $c = U$ uniforme; si $b = 0$ entonces $c = U \oplus R$, que también es uniforme. Luego c es independiente de b, y el esquema cumple **hiding**.

Binding. Doble apertura requiere $c = G(x) = G(x') \oplus R$, es decir $G(x) \oplus G(x') = R$. Sea $S = \text{Range}(G)$. Existe doble apertura si y solo si $S \cap (S \oplus R) \neq \varnothing$. Como $|S| \leq 2^n$ y el espacio total tiene tamaño 2^{3n}, un argumento de conteo da

$$\Pr_R[S \cap (S \oplus R) \neq \varnothing] \leq 2^{-n},$$

así que el esquema es **binding** (de hecho, estadísticamente) para R aleatorio.

Notas didácticas (para el alumno)

- **Estructura conceptual:** abrir = revelar x y el bit b; verificar = comprobar igualdad con $G(x)$ o $G(x) \oplus R$.
- **Por qué este método:** hiding sale de PRG $\Rightarrow G(x)$ indistinguible de uniforme.
- **Comprobaciones rápidas:** U y $U \oplus R$ tienen la misma distribución (uniforme).
- **Comprobaciones rápidas:** doble apertura $\Leftrightarrow G(x) \oplus G(x') = R$.
- **Errores comunes:** pensar que binding depende del PRG; aquí sale por conteo

del rango.

- **Errores comunes:** confundir R público con R secreto (es público).
- **Conexiones globales:** ejemplo típico de *computationally hiding* y *statistically binding*.
- **Ampliación (opcional):** si la salida fuese demasiado corta, el conteo no daría binding.

3. Considera la descripción informal del siguiente esquema de compromiso:

   - Alice elige un entero N tal que $N = pq$ para enteros primos p y q distintos.
   - Además, Alice escoge un número primo e tal que $e > N$ y $\gcd(e, \phi(N)) = 1$, y un elemento $g \in \mathbb{Z}_N^*$. La clave pública del esquema es la terna (e, N, g).
   - Alice se compromete a un mensaje $m \in \mathbb{Z}_N^*$, primero generando un elemento aleatorio $r \in \mathbb{Z}_N$ y calcula $c = m^e g^r \mod N$.
   - Para revelar el compromiso, Alice proporciona m y r para que Bob verifique que estos dos elementos corresponden al compromiso c.

   (a) Escribe los algoritmos correspondientes al esquema descrito.

   (b) Determina si este esquema cumple la propiedad de hiding.

Solución detallada

(a) Algoritmos.

- $\underline{\text{Gen}(1^n)}$: elige primos distintos p, q y define $N = pq$; calcula $\phi(N) = (p-1)(q-1)$. Elige un primo $e > N$ con $\gcd(e, \phi(N)) = 1$. Elige $g \leftarrow \mathbb{Z}_N^*$. Return ck $:= (e, N, g)$ y sk $:= (p, q)$.
- $\underline{\text{Commit}((e, N, g), m)}$: elige $r \leftarrow \mathbb{Z}_N$ uniforme y calcula

$$c := m^e \cdot g^r \mod N.$$

  Return com $:= c$ y dec $:= (m, r)$.

- $\underline{\text{Verify}((e, N, g), c, (m, r))}$: acepta si y solo si

$$c \equiv m^e g^r \pmod N.$$

(b) Hiding. El factor aleatorio g^r actúa como máscara multiplicativa. En particular, si los mensajes están en el subgrupo $\langle g \rangle$, es decir $m = g^s$, entonces

$$c = m^e g^r = g^{es+r},$$

y como r es uniforme, $es + r$ es uniforme módulo $\text{ord}(g)$ (cuando $\gcd(e, \text{ord}(g)) = 1$). En ese caso, c es uniforme en $\langle g \rangle$ e independiente de m, luego el esquema es **perfectamente hiding** para ese espacio de mensajes.
Para espacios de mensajes más generales, el hiding depende de cómo se elija g y del tamaño del subgrupo generado.

Notas didácticas (para el alumno)

- **Estructura conceptual:** compromiso $c = m^e g^r$; apertura (m, r).
- **Por qué este método:** escribir $m = g^s$ convierte c en una sola potencia g^{es+r}.
- **Comprobaciones rápidas:** r uniforme $\Rightarrow$ $es + r$ uniforme (mod el orden) si $\gcd(e, \text{ord}(g)) = 1$.
- **Comprobaciones rápidas:** Verify es solo comprobar una congruencia.
- **Errores comunes:** asumir hiding sin mirar el subgrupo que genera g (si es pequeño, puede filtrar).
- **Errores comunes:** mezclar hiding (antes de abrir) con binding (imposibilidad de doble apertura).
- **Conexiones globales:** estructura similar a Pedersen: un término fija y otro aleatoriza.
- **Ampliación (opcional):** el siguiente ejercicio explota esta estructura para una reducción a RSA.

4. **Objetivo del ejercicio:** En este ejercicio vas a demostrar que, si existe un adversario capaz de romper la propiedad *binding* del esquema de compromiso de la pregunta anterior, entonces es posible utilizarlo para descifrar un mensaje cifrado con RSA, sin conocer la clave privada.

   Situación inicial: Supón que existe un adversario A que produce un compromiso c y dos aperturas distintas $d = (m, r)$ y $d' = (m', r')$ con $m \neq m'$ que ambas verifican.

   (a) Escribe la condición matemática que debe cumplirse para que $A \diamond \mathcal{L}_{\text{binding-real}} \Rightarrow \text{true}$.

   Aplicación: Sea γ un mensaje cifrado bajo la clave pública RSA (e, N). El objetivo de B es recuperar el mensaje en claro.

   Supón que B entrega a A la terna (e, N, γ) como si fuera la clave pública del compromiso (es decir, usa $g := \gamma$) y A devuelve $(c, (m, r), (m', r'))$ cumpliendo la condición del apartado (a).

   (b) Demuestra que $\left(\frac{m'}{m}\right)^e \equiv_N \gamma^{r-r'}$.

   (c) Si $a(r - r') + be = 1$, demuestra que $\gamma = \left[\left(\frac{m'}{m}\right)^a \cdot \gamma^b\right]^e$ mód N.

   (d) Escribe en pseudocódigo los pasos que sigue B para descifrar γ usando A.

Solución detallada

En la reducción fijamos la clave pública del compromiso como (e, N, g) y tomamos $g := \gamma$.

(a) Condición para romper binding. Romper binding significa producir un mismo c y dos aperturas $(m, r) \neq (m', r')$ con $m \neq m'$ tales que

$$c \equiv m^e g^r \pmod{N} \qquad \text{y} \qquad c \equiv (m')^e g^{r'} \pmod{N}.$$

Sustituyendo $g = \gamma$:

$$c \equiv m^e \gamma^r \pmod{N} \qquad \text{y} \qquad c \equiv (m')^e \gamma^{r'} \pmod{N}.$$

(b) Deducción de $\left(\frac{m'}{m}\right)^e \equiv_N \gamma^{r-r'}$. Partimos de

$$m^e\gamma^r \equiv (m')^e\gamma^{r'} \pmod{N}.$$

Como $m, m' \in \mathbb{Z}_N^*$, podemos dividir por m^e:

$$\left(\frac{m'}{m}\right)^e \equiv \gamma^{r-r'} \pmod{N}.$$

(c) Construcción de una raíz e-èsima de γ. Elevando a la potencia a:

$$\left(\frac{m'}{m}\right)^{ae} \equiv \gamma^{a(r-r')} \pmod{N}.$$

Como $a(r-r') + be = 1$, tenemos $a(r-r') = 1 - be$, y entonces

$$\left(\frac{m'}{m}\right)^{ae} \equiv \gamma^{1-be} \pmod{N}.$$

Multiplicando por γ^{be}:

$$\left(\frac{m'}{m}\right)^{ae} \gamma^{be} \equiv \gamma \pmod{N},$$

y reagrupando:

$$\gamma \equiv \left[\left(\frac{m'}{m}\right)^a \cdot \gamma^b\right]^e \pmod{N}.$$

(d) Pseudocódigo de B.

Algoritmo $B^A(N, e, \gamma)$
1. Ejecuta $(c, (m, r), (m', r')) \leftarrow A(e, N, \gamma)$.
2. Calcula $(a, b) \leftarrow \text{ExtGCD}(r - r', e)$ tal que $a(r-r') + be = 1$.
3. Calcula $u := m' \cdot m^{-1}$ mód N.
4. Devuelve $x := u^a \cdot \gamma^b$ mód N (si $a < 0$ o $b < 0$, usa inversos modulares).

Salida: x tal que $x^e \equiv \gamma \pmod{N}$.

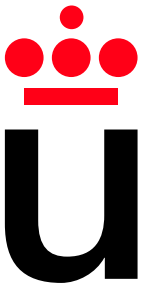
Notas didácticas (para el alumno)

- **Estructura conceptual:** doble apertura ⇒ dos ecuaciones del mismo c.
- **Por qué este método:** al dividir sale una relación entre $(m'/m)^e$ y $\gamma^{r-r'}$.
- **Comprobaciones rápidas:** asegúrate de que $m, m' \in \mathbb{Z}_N^*$ para poder invertir.
- **Comprobaciones rápidas:** con Bézout combinas exponentes para fabricar una potencia e-èsima exacta.
- **Errores comunes:** olvidar que a o b pueden ser negativos (hay que invertir módulo N).
- **Errores comunes:** no controlar el caso $\gcd(r - r', e) \neq 1$ (se evita en el planteamiento con e primo grande).

- **Conexiones globales:** esto es una reducción criptográfica: ataque a binding $\Rightarrow$ ataque a RSA.
- **Ampliación (opcional):** el patrón es el mismo que en muchas pruebas: "si hay dos aperturas, hay una ecuación algebraica explotable".

PARTE IV

Banco de Preguntas para Moodle (Categoría 7)



Universidad
Rey Juan Carlos

Banco de preguntas
CRIPTOGRAFÍA

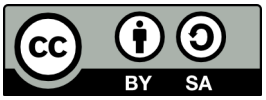
Grado en Ingeniería de la Ciberseguridad

Misael E. Marriaga

Material docente en abierto de la Universidad Rey Juan Carlos

Asignatura:	Criptografía
Titulación:	Grado en Ingeniería de la Ciberseguridad
Código:	2285007
Curso:	2025-2026

Autoría:
Misael E. Marriaga
misael.marriaga@urjc.es



©2026 Misael E. Marriaga
Algunos derechos reservados
Este documento se distribuye bajo una licencia Creative Commons, Atribución-CompartirIgual 4.0 Internacional (CC BY-SA 4.0), disponible en
https://creativecommons.org/licenses/by-sa/4.0/deed.es

Banco de Preguntas (Formato GIFT)

```
// ============================================================
// BANCO DE PREGUNTAS - CRIPTOGRAFÍA (GRADO CIBERSEGURIDAD)
// Autor: Prof. Misael E. Marriaga
// Formato: GIFT (Moodle)
// ©2026 Misael E. Marriaga
// Algunos derechos reservados
// Esta obra está bajo una Licencia Creative Commons Atribución-CompartirIgual 4.0 Internacional
// Disponible en
// https://creativecommons.org/licenses/by-sa/4.0/deed.es
// ============================================================

// --- BLOQUE 1: FUNDAMENTOS Y OTP ---

::P01:: ¿Cuál es la condición necesaria y suficiente para que un esquema de cifrado tenga Secreto Perfecto (Perfect Secrecy)? {
=La distribución de probabilidad del mensaje cifrado es independiente del mensaje original.
~Que el atacante necesite 2^128 operaciones para romperlo.
~Que la clave sea reutilizada al menos dos veces.
~Que el espacio de claves sea menor que el espacio de mensajes.
}

::P02:: En el cifrado One-Time Pad (OTP), si la clave se utiliza para cifrar dos mensajes distintos (M1 y M2), ¿qué información se revela al atacante? {
=El XOR de los dos mensajes en claro (M1 ⊕ M2).
~La clave secreta K completa.
~Solo la longitud de los mensajes.
~Nada, el OTP es incondicionalmente seguro incluso reutilizando claves.
}

::P03:: ¿Qué longitud debe tener la clave en el cifrado One-Time Pad para garantizar seguridad perfecta? {
=Al menos la misma longitud que el mensaje.
~Siempre 128 bits.
~La mitad de la longitud del mensaje.
~Cualquier longitud, siempre que sea aleatoria.
}

// --- BLOQUE 2: SIMÉTRICA (PRG, PRF, MODOS) ---

::P04:: Un Generador Pseudoaleatorio (PRG) seguro debe ser capaz de resistir: {
=El Next-Bit Test (ningún adversario eficiente puede predecir el siguiente bit con probabilidad > 1/2 + negl).
~Cualquier ataque de fuerza bruta con tiempo infinito.
~La prueba de frecuencias (tener el mismo número de ceros que de unos).
~La compresión de datos (su salida debe ser comprimible).
}

::P05:: ¿Cuál de los siguientes modos de operación NO proporciona seguridad semántica (CPA) porque es determinista? {
=ECB (Electronic Codebook)
~CBC (Cipher Block Chaining) con IV aleatorio
~CTR (Counter Mode) con IV aleatorio
~OFB (Output Feedback) con IV aleatorio
}

::P06:: En el modo CTR (Counter Mode), ¿qué propiedad permite el acceso aleatorio para descifrar un bloque concreto en medio del archivo? {
=El keystream de cada bloque se genera independientemente cifrando (Nonce ||Contador).
~El cifrado de cada bloque depende del bloque de texto cifrado anterior.
~Se utiliza una función Hash en cada bloque.
~La clave cambia en cada bloque mediante una rotación de bits.
}

::P07:: Si utilizamos un cifrado de flujo (como CTR o OFB) y un atacante modifica un bit del texto cifrado durante la transmisión (bit-flipping), ¿qué ocurre al descifrar? {
=Solo se corrompe el bit correspondiente en el texto plano; el resto del mensaje queda intacto.
~Se corrompe el bloque actual y el siguiente.
~Todo el mensaje a partir de ese punto se vuelve ilegible.
~El sistema detecta el error automáticamente y detiene el descifrado.
}

// --- BLOQUE 3: INTEGRIDAD Y HASH ---
```

::P08:: ¿Qué propiedad de seguridad NO garantiza un esquema de cifrado estándar (como AES-CBC) por sí solo? {
=Integridad del mensaje (detectar si ha sido modificado).
~Confidencialidad del mensaje.
~Imposibilidad de recuperar la clave.
~Aleatoriedad del texto cifrado.
}

::P09:: En una función Hash resistente a colisiones, ¿cuál es la complejidad aproximada para encontrar una colisión (Ataque de Cumpleaños) si la salida tiene n bits? {
=2^(n/2) operaciones.
~2^n operaciones.
~2^(n-1) operaciones.
~n^2 operaciones.
}

::P10:: ¿Cuál es la diferencia principal entre un Código de Autenticación de Mensajes (MAC) y una Firma Digital? {
=El MAC es simétrico (misma clave para generar y verificar) y la Firma es asimétrica (pública/privada).
~El MAC proporciona confidencialidad y la Firma no.
~El MAC es mucho más lento que la Firma Digital.
~La Firma Digital requiere un canal seguro previo para compartir la clave.
}

::P11:: ¿Por qué la construcción Merkle-Damgård (usada en SHA-2) añade el relleno (padding) conteniendo la longitud del mensaje al final? {
=Para prevenir ataques de extensión de longitud (Length Extension Attacks).
~Para que la longitud total sea múltiplo de 8.
~Para ocultar la longitud real del mensaje al atacante.
~Para aumentar la entropía del mensaje.
}

// --- BLOQUE 4: ASIMÉTRICA Y TEORÍA DE NÚMEROS ---

::P12:: En el protocolo de intercambio de claves Diffie-Hellman, la seguridad se basa en la dificultad de resolver el problema del: {
=Logaritmo Discreto (DLOG) o Diffie-Hellman Computacional (CDH).
~Factorización de enteros grandes.
~Mochila (Knapsack problem).
~Reticulados (Lattices).
}

::P13:: En el cifrado RSA "de libro de texto" (sin padding), ¿qué propiedad indeseable permite a un atacante crear un cifrado válido de (m1 * m2) a partir de los cifrados de m1 y m2? {
=Maleabilidad (propiedad homomórfica multiplicativa).
~Falta de aleatoriedad.
~Determinismo.
~Todas las anteriores son correctas.
}

::P14:: En un sistema RSA, si la clave pública es (N, e), ¿cuál es la relación matemática que debe cumplir la clave privada d? {
=e * d ≡ 1 (mod ϕ(N))
~e * d ≡ 0 (mod N)
~e + d = N
~d = e^N (mod ϕ(N))
}

::P15:: ¿Qué ventaja principal tiene la Criptografía de Curva Elíptica (ECC) frente a RSA? {
=Ofrece la misma seguridad con claves mucho más cortas (ej. 256 bits vs 3072 bits).
~Es resistente a ordenadores cuánticos (Post-Quantum).
~No requiere operaciones modulares.
~Es más fácil de explicar matemáticamente.
}

// --- BLOQUE 5: FIRMAS Y CONCEPTOS AVANZADOS ---

::P16:: En un esquema de Firma Digital RSA, el proceso de firma consiste en: {
=Aplicar la clave PRIVADA al hash del mensaje: s = H(m)^d mod N.
~Aplicar la clave PÚBLICA al hash del mensaje: s = H(m)^e mod N.
~Cifrar el mensaje con la clave pública del receptor.
~Cifrar el mensaje con una clave simétrica y firmar la clave.
}

::P17:: ¿Qué ataque es posible si un esquema de firma digital es determinista y no utiliza una función hash sobre el mensaje antes de firmar? {
=Falsificación existencial (Existential Forgery).
~Recuperación total de la clave privada.
~Ataque de repetición (Replay attack).

~Colisión de claves.
}

::P18:: En el contexto de Compartición de Secretos de Shamir (t, n), ¿cuántas partes se necesitan como mínimo para reconstruir el secreto? {
=t partes.
~n partes.
~t-1 partes.
~n/2 partes.
}

::P19:: ¿Qué garantiza la propiedad de "Binding" (Vinculación) en un esquema de compromiso? {
=Que el emisor no puede cambiar el valor comprometido después de haber enviado el compromiso.
~Que el receptor no puede ver el valor comprometido antes de la apertura.
~Que el compromiso siempre tiene longitud fija.
~Que el esquema utiliza criptografía asimétrica.
}

::P20:: El ataque "Man-in-the-Middle" en Diffie-Hellman es posible si: {
=No existe autenticación de las claves públicas intercambiadas.
~Los números primos elegidos son demasiado pequeños.
~Se reutilizan los exponentes privados.
~El canal de comunicación es ruidoso.
}

PARTE V

Software utilizado en la asignatura (Categoría 6)

Ficha del software

Campo	Contenido
Nombre	PyCrypto-Edu: Librería Educativa de Criptografía
Versión	1.0 (Curso 2025–2026)
Descripción	Conjunto de scripts en Python que implementan primitivas criptográficas básicas (cifrado César, Vigenère, RSA simplificado, Diffie–Hellman) con fines estrictamente didácticos, priorizando la legibilidad del código sobre la eficiencia o la seguridad en entornos de producción.
Autor	Misael E. Marriaga (Universidad Rey Juan Carlos)
Licencia	GNU General Public License v3.0 (GPL-3.0)
Lenguaje	Python
Requerimientos	Ver archivo requirements.txt incluido en el paquete
Uso docente	Laboratorios prácticos de la asignatura de Criptografía
Repositorio de desarrollo	https://github.com/mmarriag-collab/pycrypto-edu-urjc
Archivo permanente	Software Heritage
SWHID	swh:1:dir:214c5cb54d6e10982e50f079975c606e77fb6bde
URL Software Heritage	https://archive.softwareheritage.org/swh:1:dir:214c5cb54d6e10982e50f079975c606e77fb6bde
Repositorio institucional	BURJC Digital
Instrucciones de uso	Véase el archivo README.md incluido en el paquete