Universidad Nacional Autónoma de México
Facultad de Ciencias - Ciencias de la Computación
Criptografía y seguridad 2018-1

Práctica 2

1. Modos de operación

En los cifradores por bloques se utilizan los modos de operación para poder cifrar y descifrar mensajes de tamaño arbitrario, en vez de restringirse al tamaño del bloque. Por ejemplo, podemos tener un algoritmo que trabaja con bloques de 16 bytes pero queremos cifrar mensajes de 100 megabytes o más.

Sea ℓ el tamaño de bloque y k la llave que se usará. La forma ingenua de encriptar mensajes de tamaño $> \ell$ es dividir el mensaje en bloques de tamaño ℓ y aplicar $\mathsf{Enc}_k(\cdot)$ a cada bloque. A este modo de operación se le llama Electronic Codebook (ECB) y nunca debe usarse, ya que si en el mensaje original aparecen algunos bloques iguales, estos se transformarán en bloques que serán iguales, y esto proporciona información sobre el mensaje original.

Existen muchos modos de operación, cada uno con ciertas ventajas y desventajas. Los siguientes diagramas muestran algunos de los más comunes.

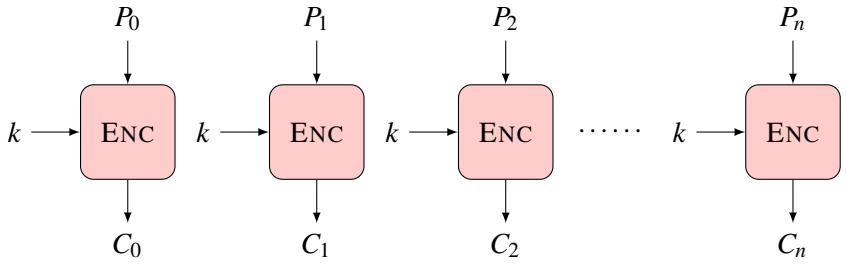

Figura 1: Electronic Codebook (ECB)

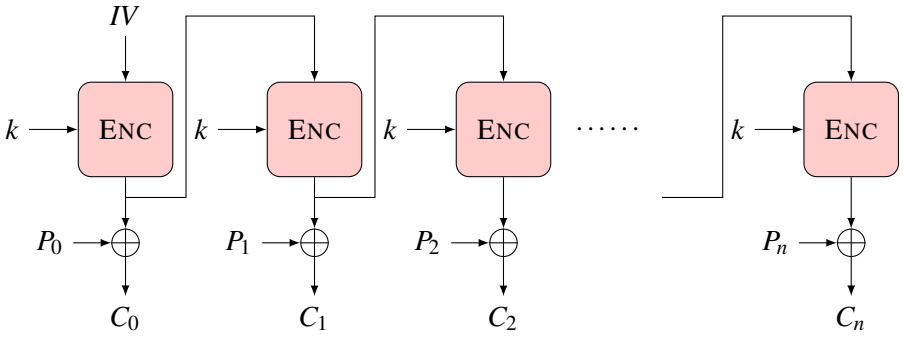

Figura 2: Output Feedback (OFB)

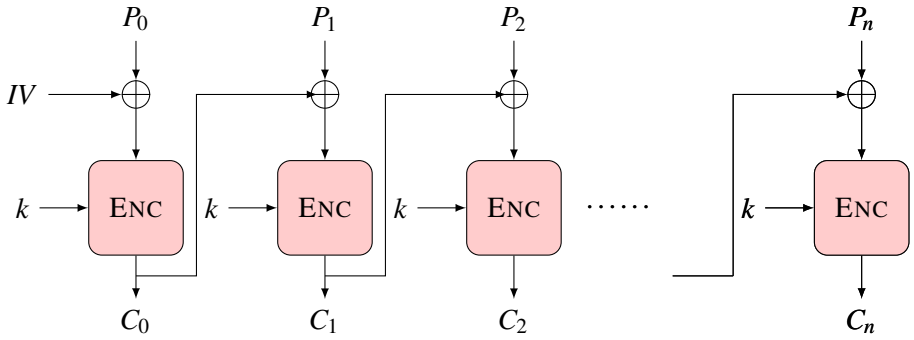

Figura 3: Cipher Block Chaining (CBC)

1.1. Padding

Cuando el tamaño del mensaje no es múltiplo del tamaño de bloque, necesitamos agregar un relleno para poder aplicar el cifrado a todos los bloques, a este proceso se le llama *padding* (también a la cadena que se agrega se le llama así). Por ejemplo, sea $m =$ `COMPUTADORA` y el tamaño de bloque $\ell = 5$. Como $|m| = 11$, hay que agregar cuatro símbolos para tener un múltiplo de ℓ y así poder partir m en tres bloques: `COMPU TADOR AXXXX`, donde `XXXX` es el relleno o padding.

Para este ejercicio se usará el padding que consiste en agregar un byte `00` seguido de bytes `ff` necesarios hasta obtener el tamaño de bloque. Recuerda que el padding debe aplicarse siempre, aun si el mensaje original tiene tamaño múltiplo de ℓ.

1.2. Programa

1. Busca una implementación de AES. Por ejemplo, PyCrypto para Python, javax.crypto para Java y OpenSSL para C. Esta implementación debe permitirte usar el modo ECB para cifrar de forma individual un bloque. En particular se usarán solo llaves de 16 bytes (128 bits).

2. Implementa el cifrado y descifrado usando los modos OFB y CBC. Para cada modo usarás un vector inicial *IV* de 16 bytes, que será obtenido de `/dev/urandom`. El vector inicial siempre se guardará al inicio del mensaje cifrado.

3. El programa se ejecutará de la siguiente forma

   ```
   $ cifrador [e|d] [cbc|ofb] archivo_llave archivo_claro
   ```

   donde `[e|d]` es para indicar si se quiere cifrar o descifrar, `[cbc|ofb]` indica el modo de operación, `archivo_llave` contiene una llave de 16 bytes y `archivo_claro` es el nombre del archivo que será cifrado (que pesará menos de 1 gigabyte).

4. (Extra) El programa puede aceptar cualquier cadena de 16 bytes como llave, pero esta llave no es lo mismo que una contraseña como la que se usa en correo electrónico o Facebook. En el ejercicio 2.3 se trabaja con una *función de derivación de llaves* (KDF), incorpora una a tu programa para poder cifrar usando contraseñas arbitrarias.

2. Funciones hash

Una función hash criptográfica es una función que toma como entrada cadenas de bits de cualquier tamaño y devuelve una cadena de tamaño fijo n, es decir,

$$H\colon \{0,1\}^* \to \{0,1\}^n$$

y además satisface las siguientes propiedades:

Resistencia a preimágenes. Sea $y = H(x)$ el valor hash de algún mensaje x. Si solo conocemos y, es difícil encontrar un valor x' tal que $H(x') = y$ (x' puede ser x o distinto).

Resistencia a segunda preimagen. Dado un mensaje x, es difícil encontrar un x' distinto tal que $H(x) = H(x')$.

Resistencia a colisiones. Es difícil encontrar dos mensajes distintos x y y que tengan el mismo valor, es decir, tales que $H(x) = H(y)$. A una pareja de valores así se le llama colisión.

Algunas funciones hash conocidas son MD5, SHA1, la familia SHA2 (de varios tamaños) y BLAKE. Por ejemplo, SHA1 produce un valor hash de 160 bits ($n = 160$, 20 bytes),

$$\text{SHA1}(\text{“}\texttt{abc}\text{”}) = \texttt{a9993e364706816aba3e25717850c26c9cd0d89d}$$
$$\text{SHA1}(\text{cadena vacía}) = \texttt{da39a3ee5e6b4b0d3255bfef95601890afd80709}.$$

2.1. Contraseñas con funciones hash

Al guardar una cuenta de usuario en una base de datos, nunca se almacenan las contraseñas directamente. Una forma de guardar esta información es usar una función hash para aplicarséla a las contraseñas y guardar el valor hash.

Por ejemplo, usemos SHA1. Cuando un usuario escoge la contraseña “`abc`”, en la base de datos se guardará el valor `a9993e364706816aba3e25717850c26c9cd0d89d`. Así, cuando el usuario introduzca una contraseña x se calcula SHA1(x) y se compara con el valor guardado en la base de datos, si coinciden los valores quiere decir que se introdujo la contraseña correcta.

Ahora supongamos que alguien no autorizado tiene acceso a la base de datos y puede ver los valores hash de las contraseñas. En principio no puede saber las contraseñas, solo puede ver el valor hash, y como la función hash es resistente a preimágenes no se pueden encontrar las contraseñas correspondientes. Un problema es que si dos usuarios escogieron la misma contraseña al registrarse, en la base de datos ambos usuarios tendrán el mismo hash, y debido a esto, si el intruso puede adivinar una contraseña, de inmediato sabrá la contraseña de todos los usuarios que escogieron la misma.

Para tratar de mitigar este problema, a una contraseña se le agrega una pequeña cadena aleatoria conocida como *salt*, de forma que dos usuarios con la misma contraseña obtendrán valores hash distintos. Esto es: $H(p\,||\,salt_1) \neq H(p\,||\,salt_2)$, donde p es la contraseña y el símbolo $||$ denota concatenación. El valor salt se almacena junto con el hash para después poder recuperar $H(p\,||\,salt)$.

Este método hace que todos los usuarios de la base de datos tengan un valor hash de contraseña distinto, aun cuando la contraseña se repita. Aun así, hay una forma simple de intentar recuperar las contraseñas.

2.2. Echarle sal no basta

El archivo `passwords-salt` contiene una lista de contraseñas obtenidas de una base de datos. Fueron creadas usando la siguiente función

```
function pwd_hash(salt, password)
    hash_val = sha256(password || salt)
    return '$' || salt || '$' || base64(hash_val)
```

(`||` es concatenación)

Las contraseñas originales aparecen en la lista `common-passwords.txt`. Encuentra cuáles son y guárdalas en un archivo llamado `passwords-originales`. Toma el tiempo que le tardó a tu programa encontrar todas las contraseñas, además anexa tu programa.

2.3. Función de derivación de llaves

Investiga qué es una función de derivación de llaves, en particular una conocida como PBKDF2.

De las primeras 300 contraseñas del archivo `common-passwords` se escogió una, luego se le aplicó PBKDF2 con la cadena salt `4H7PSmnC1B` usando SHA256, 200000 iteraciones y salida de 32 bytes. El resultado en base64 es el siguiente

`NPyyv+PZd+znB1Mva7MYwRg0b0oDQUX+0BDSFv3uod4=`

Encuentra la contraseña. Toma el tiempo que le tomó a tu programa encontrarla y compara con el tiempo del ejercicio anterior. Anexa el programa que usaste.

Extras

Los siguientes ejercicios son opcionales y sirven para subir calificación tanto en la primera práctica como en esta (donde sean necesarios).

3. Una función hash insegura (5 puntos)

Podemos usar DES para construir una función hash para mensajes de tamaño fijo. Definimos $h\colon \{0,1\}^{112} \to \{0,1\}^{64}$ como $h(x_1||x_2) = \text{DES}_{x_1}(\text{DES}_{x_2}(0^{64}))$ donde $|x_1| = |x_2| = 56$. Es decir, h acepta mensajes m de 112 bits (14 bytes), se divide por la mitad para obtener $m = x_1x_2$, y usando DES se encripta un bloque de bytes `0x00`, primero se usa x_2 como llave, luego se vuelve a aplicar DES pero con llave x_1; el resultado es el valor de $h(m)$. (*Nota:* DES usa llaves de 64 bits, pero 8 bits son de paridad, por lo que solamente sirven 56 bits. Algunas implementaciones –como PyCrypto– toman llaves de 64 bits pero ignoran el último bit de cada byte.)

a) Una *llave débil* para DES es una llave k tal que $\text{DES}_k(\text{DES}_k(m)) = m$. Usando llaves débiles encuentra una colisión para h.

b) Dado un mensaje y describe cómo hallar una preimagen usando 2^{56} operaciones, es decir, encontrar x_1, x_2 tales que $h(x_1||x_2) = y$.

c) Usando la paradoja del cumpleaños describe cómo hallar una preimagen con alta probabilidad usando 2^{32} operaciones. Justifica.

d) Basándote en la respuesta anterior encuentra m tal que $h(m) = y$, donde y empieza con los bytes `00 de ad`. Nota que esto es como encontrar una preimagen, pero fijándonos solamente en los primeros bytes.

La minería de bitcoins consiste en encontrar mensajes m tales que $\text{SHA256}(\text{SHA256}(m)) = y$, donde los primeros r bytes de y son ceros (r es un parámetro que va cambiando).

e) Escribe tus respuestas en un archivo de texto y también anexa tu programa.

4. Entrega

- La práctica puede hacerse entre dos personas o de forma individual.
- Organiza tus archivos en un directorio y comprímelo en un zip. Tanto el directorio como el archivo zip tendrán un nombre de la forma `Practica2_Lopez` o `Practica2_Lopez_Juarez`, que indican el apellido paterno de una o dos personas, respectivamente.
- Sube tu archivo zip en el siguiente formulario: `https://goo.gl/forms/6sxpGkR2HwyEy6L33`
- Fecha límite: 29 de octubre a cualquier hora.
- No se aceptará por correo.