MOOC de Criptología Matemática. Funciones de Resumen Digital (HASH)

Leandro Marín

Módulo I. Sesión 5.
Dificultad Alta

Definición

- Una función de resumen digital (hash) es una función h que aplicada a un mensaje m nos da un valor numérico $h(m)$ de longitud constante.

Definición

- Una función de resumen digital (hash) es una función h que aplicada a un mensaje m nos da un valor numérico $h(m)$ de longitud constante.
- La longitud de $h(m)$ suele ser un número entre 160 y 512 bits.

Definición

- Una función de resumen digital (hash) es una función h que aplicada a un mensaje m nos da un valor numérico $h(m)$ de longitud constante.
- La longitud de $h(m)$ suele ser un número entre 160 y 512 bits.
- La longitud del mensaje m puede ser cualquiera.

Definición

- Una función de resumen digital (hash) es una función h que aplicada a un mensaje m nos da un valor numérico $h(m)$ de longitud constante.
- La longitud de $h(m)$ suele ser un número entre 160 y 512 bits.
- La longitud del mensaje m puede ser cualquiera.
- A diferencia de los sistemas de cifrado, las funciones de resumen digital no dependen de la elección de claves.

Propiedades

- Sea h una función de resumen digital con salida de longitud n bits. La función h debe cumplir las siguientes propiedades:

Propiedades

- Sea h una función de resumen digital con salida de longitud n bits. La función h debe cumplir las siguientes propiedades:
  - Resistencia de preimagen: Dada una salida x debe ser difícil encontrar un mensaje m tal que $h(m) = x$. El problema debe tener una dificultad equivalente a probar todas las combinaciones posibles, es decir 2^n.

Propiedades

- Sea h una función de resumen digital con salida de longitud n bits. La función h debe cumplir las siguientes propiedades:
  - Resistencia de preimagen: Dada una salida x debe ser difícil encontrar un mensaje m tal que $h(m) = x$. El problema debe tener una dificultad equivalente a probar todas las combinaciones posibles, es decir 2^n.
  - Segunda resistencia en preimagen: Dado un mensaje m y una salida x tal que $h(m) = x$ debe ser difícil encontrar otro mensaje m' tal que $h(m') = x$. El problema debe tener una dificultad equivalente a probar todas las combinaciones posibles, es decir 2^n.

Propiedades

- Sea h una función de resumen digital con salida de longitud n bits. La función h debe cumplir las siguientes propiedades:
  - Resistencia de preimagen: Dada una salida x debe ser difícil encontrar un mensaje m tal que $h(m) = x$. El problema debe tener una dificultad equivalente a probar todas las combinaciones posibles, es decir 2^n.
  - Segunda resistencia en preimagen: Dado un mensaje m y una salida x tal que $h(m) = x$ debe ser difícil encontrar otro mensaje m' tal que $h(m') = x$. El problema debe tener una dificultad equivalente a probar todas las combinaciones posibles, es decir 2^n.
  - Resistencia de colisión: Debe ser difícil encontrar mensajes m y m' tales que $h(m) = h(m')$. La dificultad de este problema debe ser del orden $2^{n/2}$, que es la probabilidad estadística de encontrarlos al azar, tal y como muestra la paradoja del cumpleaños.

Modelo en rondas

- El proceso que se utiliza para definir funciones de resumen digital es similar al que se hace para DES y AES.

Modelo en rondas

- El proceso que se utiliza para definir funciones de resumen digital es similar al que se hace para DES y AES.
- El mensaje m se parte en trozos y se definen una serie de rondas en las cuales se van procesando cada uno de los trozos hasta terminar de procesar el mensaje.

Modelo en rondas

- El proceso que se utiliza para definir funciones de resumen digital es similar al que se hace para DES y AES.
- El mensaje m se parte en trozos y se definen una serie de rondas en las cuales se van procesando cada uno de los trozos hasta terminar de procesar el mensaje.
- Estas rondas están definidas mediante funciones y constantes determinadas en la especificación del algoritmo.

Funciones

- Cada una de las rondas en las que se suele descomponer el algoritmo aplica una función.

Funciones

- Cada una de las rondas en las que se suele descomponer el algoritmo aplica una función.
- Estas funciones se repetirán de forma cíclica para ajustarse a la longitud total del mensaje.

Funciones

- Cada una de las rondas en las que se suele descomponer el algoritmo aplica una función.
- Estas funciones se repetirán de forma cíclica para ajustarse a la longitud total del mensaje.
- Suelen estar definidas en términos de operadores lógicos, operadores aritméticos y rotaciones.

Constantes

- Los algoritmos suelen disponer de constantes.

Constantes

- Los algoritmos suelen disponer de constantes.
- Estas constantes pueden depender de la ronda en la que nos encontremos, tal y como sucedía con las funciones.

Constantes

- Los algoritmos suelen disponer de constantes.
- Estas constantes pueden depender de la ronda en la que nos encontremos, tal y como sucedía con las funciones.
- Suelen utilizarse en las funciones junto con los valores previamente procesados.

Proceso de Relleno

- Los mensajes pueden tener longitud variable, sin embargo debemos partirlos en trozos que deben tener longitud fija.

Proceso de Relleno

- Los mensajes pueden tener longitud variable, sin embargo debemos partirlos en trozos que deben tener longitud fija.
- Para conseguir esto se suele rellenar el mensaje con información adicional hasta que la longitud sea un múltiplo exacto de la longitud de los trozos que debemos procesar.

Proceso de Relleno

- Los mensajes pueden tener longitud variable, sin embargo debemos partirlos en trozos que deben tener longitud fija.
- Para conseguir esto se suele rellenar el mensaje con información adicional hasta que la longitud sea un múltiplo exacto de la longitud de los trozos que debemos procesar.
- El algoritmo describirá cómo hacer este preoceso de relleno previo al cálculo del resumen digital.

Proceso de Relleno

- Los mensajes pueden tener longitud variable, sin embargo debemos partirlos en trozos que deben tener longitud fija.
- Para conseguir esto se suele rellenar el mensaje con información adicional hasta que la longitud sea un múltiplo exacto de la longitud de los trozos que debemos procesar.
- El algoritmo describirá cómo hacer este preoceso de relleno previo al cálculo del resumen digital.
- Habitualmente se añadirán ceros y una indicación de la longitud del mensaje original.

Valores Iniciales

- En cada ronda se suelen combinar los valores previamente calculados junto con el trozo de mensaje correspondiente a la ronda.

Valores Iniciales

- En cada ronda se suelen combinar los valores previamente calculados junto con el trozo de mensaje correspondiente a la ronda.
- Para la primera ronda debemos partir de algunos valores iniciales.

Valores Iniciales

- En cada ronda se suelen combinar los valores previamente calculados junto con el trozo de mensaje correspondiente a la ronda.
- Para la primera ronda debemos partir de algunos valores iniciales.
- Estos vendrán determinados en la definición del algoritmo.

Introducción

- En la sección anterior hemos descrito la estructura general de las funciones de resumen digital.

Introducción

- En la sección anterior hemos descrito la estructura general de las funciones de resumen digital.
- Vamos ahora a ver detalladamente el caso de la función SHA-1.

Introducción

- En la sección anterior hemos descrito la estructura general de las funciones de resumen digital.
- Vamos ahora a ver detalladamente el caso de la función SHA-1.
- Aunque actualmente esta función ya no se considera totalmente segura, sigue siendo parte de muchos estándares.

Introducción

- En la sección anterior hemos descrito la estructura general de las funciones de resumen digital.
- Vamos ahora a ver detalladamente el caso de la función SHA-1.
- Aunque actualmente esta función ya no se considera totalmente segura, sigue siendo parte de muchos estándares.
- Iremos viendo cada uno de sus elemento junto con una implementación de dicha función en sage.

Introducción

- En la sección anterior hemos descrito la estructura general de las funciones de resumen digital.
- Vamos ahora a ver detalladamente el caso de la función SHA-1.
- Aunque actualmente esta función ya no se considera totalmente segura, sigue siendo parte de muchos estándares.
- Iremos viendo cada uno de sus elemento junto con una implementación de dicha función en sage.
- Seguiremos detalladamente el estándar FIPS que la define junto con sus notaciones. Este estándar se puede consultar en `http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf`

Relleno del Mensaje

- En SHA-1 se procean bloques de 512 bits, por lo que debemos obtener una cadena inicial con un número de bits múltiplo de 512.

Relleno del Mensaje

- En SHA-1 se procean bloques de 512 bits, por lo que debemos obtener una cadena inicial con un número de bits múltiplo de 512.
- Para ello se pone un 1 seguido de tantos ceros como sea necesario para que falten sólo 64 bits para terminar el bloque de 512 bits.

Relleno del Mensaje

- En SHA-1 se procean bloques de 512 bits, por lo que debemos obtener una cadena inicial con un número de bits múltiplo de 512.
- Para ello se pone un 1 seguido de tantos ceros como sea necesario para que falten sólo 64 bits para terminar el bloque de 512 bits.
- Esos últimos 64 bits se utilizarán para escribir en binario la longitud en bits del mensaje original.

Implementación del Proceso de Relleno

```
def relleno(B):
  l = len(B)
  B = [1]+B
  while((len(B)+64)%512!=0):
    B = [0]+B
  L = ZZ(l).digits(2,None,64)
  return L+B
```

Descomposición en Bloques

- Una vez rellenado el mensaje inicial hasta conseguir una lista de bits con una longitud múltiplo de 512 procederemos como sigue:

Descomposición en Bloques

- Una vez rellenado el mensaje inicial hasta conseguir una lista de bits con una longitud múltiplo de 512 procederemos como sigue:
- Partiremos la lista en trozos de 32 bits.

Descomposición en Bloques

- Una vez rellenado el mensaje inicial hasta conseguir una lista de bits con una longitud múltiplo de 512 procederemos como sigue:
- Partiremos la lista en trozos de 32 bits.
- Cada ronda utilizará 16 bloques de 32 bits (512 bits en total).

Descomposición en Bloques

- Una vez rellenado el mensaje inicial hasta conseguir una lista de bits con una longitud múltiplo de 512 procederemos como sigue:
- Partiremos la lista en trozos de 32 bits.
- Cada ronda utilizará 16 bloques de 32 bits (512 bits en total).
- Estos bloques de 32 bits se manejarán como números (no como listas de bits).

Implementación de la Descomposición en Bloques

```
def bloques(cadena):
  B = []
  for x in cadena:
    X = ZZ(ord(x)).digits(2,None,8)
    B = X+B
  B = relleno(B)
  M = ZZ(B,2).digits(2^32)
  M.reverse()
  return M
```

Notemos que la función `ord(x)` nos da el valor ASCII de cada carácter de la cadena.

Funciones

- Este algoritmo tiene tres funciones:

Funciones

- Este algoritmo tiene tres funciones:
  - $Ch(x, y, z) = (x \texttt{ and } y) \texttt{ xor } (\texttt{not}(x) \texttt{ and } z)$

Funciones

- Este algoritmo tiene tres funciones:
  - $Ch(x, y, z) = (x \texttt{ and } y) \texttt{ xor } (\texttt{not}(x) \texttt{ and } z)$
  - $Parity(x, y, z) = x \texttt{ xor } y \texttt{ xor } z$

Funciones

- Este algoritmo tiene tres funciones:
  - $Ch(x, y, z) = (x \texttt{ and } y) \texttt{ xor } (\texttt{not}(x) \texttt{ and } z)$
  - $Parity(x, y, z) = x \texttt{ xor } y \texttt{ xor } z$
  - $Maj(x, y, z) = (x \texttt{ and } y) \texttt{ xor } (x \texttt{ and } z) \texttt{ xor } (y \texttt{ and } z)$

Funciones

- Este algoritmo tiene tres funciones:
  - $Ch(x, y, z) = (x \texttt{ and } y) \texttt{ xor } (\texttt{not}(x) \texttt{ and } z)$
  - $Parity(x, y, z) = x \texttt{ xor } y \texttt{ xor } z$
  - $Maj(x, y, z) = (x \texttt{ and } y) \texttt{ xor } (x \texttt{ and } z) \texttt{ xor } (y \texttt{ and } z)$
- Dependiendo de un parámetro t situado entre 0 y 60 utilizaremos una u otra.

Implementación de las Funciones

```
def Ch(x,y,z):
  return (x & y) ^^ ((x ^^ 0xffffffff) & z)

def Parity(x,y,z):
  return x^^y^^z

def Maj(x,y,z):
  return (x & y) ^^ (x & z) ^^ (y & z)

def f(t,x,y,z):
  if t <= 19:
    return Ch(x,y,z)
  elif t <= 39:
    return Parity(x,y,z)
  elif t <= 59:
    return Maj(x,y,z)
  else:
    return Parity(x,y,z)
```

Rotación a la Izquierda

- Adicionalmente tendremos que hacer rotaciones hacia la izquierda de un cierto número de bits sobre palabras de 32 bits.

Rotación a la Izquierda

- Adicionalmente tendremos que hacer rotaciones hacia la izquierda de un cierto número de bits sobre palabras de 32 bits.
- Estas rotaciones se pueden implementar mediante particiones de listas, tal y como vimos en un tema anterior.

Implementación de la Rotación a la Izquierda

```
def rotar(x,k):
  X = ZZ(x).digits(2,None,32)
  X.reverse()
  X = X[k:]+X[:k]
  X.reverse()
  return ZZ(X,2)
```

Constantes

- En SHA-1 disponemos de 4 constantes que también se repetirán dependiendo de un parámetro t siguiendo un proceso similar al que se hacía para las funciones.

Constantes

- En SHA-1 disponemos de 4 constantes que también se repetirán dependiendo de un parámetro t siguiendo un proceso similar al que se hacía para las funciones.
- Se puede implementar del siguiente modo:

```
def K(t):
  if t <= 19:
    return 0x5a827999
  elif t <= 39:
    return 0x6ed9eba1
  elif t <= 59:
    return 0x8f1bbcdc
  else:
    return 0xca62c1d6
```

Valores Iniciales

- El algoritmo utilizará 5 listas marcadas desde H0 hasta H4.

Valores Iniciales

- El algoritmo utilizará 5 listas marcadas desde H0 hasta H4.
- Dichas listas se incrementarán en un valor para cada ronda.

Valores Iniciales

- El algoritmo utilizará 5 listas marcadas desde H0 hasta H4.
- Dichas listas se incrementarán en un valor para cada ronda.
- Los valores iniciales serán los siguientes:

```
H0 = [0x67452301]
H1 = [0xefcdab89]
H2 = [0x98badcfe]
H3 = [0x10325476]
H4 = [0xc3d2e1f0]
```

Rondas

- Se procederá a leer bloques de 512 bits en forma de 16 boques con números de 32 bits.

Rondas

- Se procederá a leer bloques de 512 bits en forma de 16 boques con números de 32 bits.
- Esos valores se denotarán `W[0]`, ... , `W[15]`

Rondas

- Se procederá a leer bloques de 512 bits en forma de 16 boques con números de 32 bits.
- Esos valores se denotarán `W[0], ... , W[15]`
- Se definirán los valores `W[16], ... , W[79]` de forma iterativa con la fórmula

```
W[t] = rotar(W[t-3]^^W[t-8]^^W[t-14]^^W[t-16],1)
```

Rondas

- Se procederá a leer bloques de 512 bits en forma de 16 boques con números de 32 bits.
- Esos valores se denotarán `W[0]`, ... , `W[15]`
- Se definirán los valores `W[16]`, ... , `W[79]` de forma iterativa con la fórmula

```
W[t] = rotar(W[t-3]^^W[t-8]^^W[t-14]^^W[t-16],1)
```

- Una vez definidos esos 80 valores, se procederá a aplicar las 80 funciones y constantes definidas anteriormente.

Rondas

- Se procederá a leer bloques de 512 bits en forma de 16 boques con números de 32 bits.
- Esos valores se denotarán `W[0]`, ... , `W[15]`
- Se definirán los valores `W[16]`, ... , `W[79]` de forma iterativa con la fórmula

```
W[t] = rotar(W[t-3]^^W[t-8]^^W[t-14]^^W[t-16],1)
```

- Una vez definidos esos 80 valores, se procederá a aplicar las 80 funciones y constantes definidas anteriormente.
- Hechas esas 80 transformaciones, se añadirán los resultados al final de las listas H0, H1, H2, H3 y H4.

Implementación de las Rondas

```
for i in range(len(M)/16):
  W = [M[t+16*i] for t in range(16)]
  for t in range(16,80):
    W.append(rotar(W[t-3]^^W[t-8]^^W[t-14]^^W[t-16],1))
  a = H0[-1]
  b = H1[-1]
  c = H2[-1]
  d = H3[-1]
  e = H4[-1]
  for t in range(80):
    T = (rotar(a,5)+f(t,b,c,d)+e+K(t)+W[t]) & 0xffffffff
    e = d
    d = c
    c = rotar(b,30)
    b = a
    a = T
  H0.append((a+H0[-1]) & 0xffffffff)
  H1.append((b+H1[-1]) & 0xffffffff)
  H2.append((c+H2[-1]) & 0xffffffff)
  H3.append((d+H3[-1]) & 0xffffffff)
  H4.append((e+H4[-1]) & 0xffffffff)
```

Obtención del Resultado Final

- Una vez procesado todo el mensaje, se calculará el resultado final juntando los últimos valores introducidos en las listas H0,...,H4 en un único número de 160 bits.

Obtención del Resultado Final

- Una vez procesado todo el mensaje, se calculará el resultado final juntando los últimos valores introducidos en las listas H0,...,H4 en un único número de 160 bits.
- Se implementará como sigue:

```
h0 = ZZ(H0[-1]).digits(2,None,32)
h1 = ZZ(H1[-1]).digits(2,None,32)
h2 = ZZ(H2[-1]).digits(2,None,32)
h3 = ZZ(H3[-1]).digits(2,None,32)
h4 = ZZ(H4[-1]).digits(2,None,32)
hh = h4+h3+h2+h1+h0
return ZZ(hh,2)
```

Ejemplos de Ejecución

- Podemos calcular algunos valores conocidos, como por ejemplo la cadena vacía, que nos dará el resultado hexadecimal:

```
sage: print hex(sha1(""))
da39a3ee5e6b4b0d3255bfef95601890afd80709
```

Ejemplos de Ejecución

- Podemos calcular algunos valores conocidos, como por ejemplo la cadena vacía, que nos dará el resultado hexadecimal:

```
sage: print hex(sha1(""))
da39a3ee5e6b4b0d3255bfef95601890afd80709
```

- O la cadena
  `The quick brown fox jumps over the lazy dog` que nos dará:

```
print hex(sha1("The quick brown fox jumps over the
                                lazy dog"))
2fd4e1c67a2d28fced849ee1bb76e7391b93eb12
```

Código Completo I

```
def Ch(x,y,z):
  return (x & y) ^^ ((x ^^ 0xffffffff) & z)

def Parity(x,y,z):
  return x^^y^^z

def Maj(x,y,z):
  return (x & y) ^^ (x & z) ^^ (y & z)

def rotar(x,k):
  X = ZZ(x).digits(2,None,32)
  X.reverse()
  X = X[k:]+X[:k]
  X.reverse()
  return ZZ(X,2)

def f(t,x,y,z):
  if t <= 19:
    return Ch(x,y,z)
  elif t <= 39:
    return Parity(x,y,z)
  elif t <= 59:
    return Maj(x,y,z)
```

Código Completo II

```
    else:
      return Parity(x,y,z)

def K(t):
    if t <= 19:
      return 0x5a827999
    elif t <= 39:
      return 0x6ed9eba1
    elif t <= 59:
      return 0x8f1bbcdc
    else:
      return 0xca62c1d6

def relleno(B):
    l = len(B)
    B = [1]+B
    while((len(B)+64)%512!=0):
      B = [0]+B
    L = ZZ(l).digits(2,None,64)
    return L+B

def bloques(cadena):
    B = []
    for x in cadena:
```

Código Completo III

```
    X = ZZ(ord(x)).digits(2,None,8)
    B = X+B
  B = relleno(B)
  M = ZZ(B,2).digits(2^32)
  M.reverse()
  return M

def sha1(cadena):
  M = bloques(cadena)
  H0 = [0x67452301]
  H1 = [0xefcdab89]
  H2 = [0x98badcfe]
  H3 = [0x10325476]
  H4 = [0xc3d2e1f0]
  for i in range(len(M)/16):
    W = [M[t+16*i] for t in range(16)]
    for t in range(16,80):
      W.append(rotar(W[t-3]^^W[t-8]^^W[t-14]^^W[t-16],1))
    a = H0[-1]
    b = H1[-1]
    c = H2[-1]
    d = H3[-1]
    e = H4[-1]
    for t in range(80):
```

Código Completo IV

```
        T = (rotar(a,5)+f(t,b,c,d)+e+K(t)+W[t]) & 0xffffffff
        e = d
        d = c
        c = rotar(b,30)
        b = a
        a = T
      H0.append((a+H0[-1]) & 0xffffffff)
      H1.append((b+H1[-1]) & 0xffffffff)
      H2.append((c+H2[-1]) & 0xffffffff)
      H3.append((d+H3[-1]) & 0xffffffff)
      H4.append((e+H4[-1]) & 0xffffffff)
    h0 = ZZ(H0[-1]).digits(2,None,32)
    h1 = ZZ(H1[-1]).digits(2,None,32)
    h2 = ZZ(H2[-1]).digits(2,None,32)
    h3 = ZZ(H3[-1]).digits(2,None,32)
    h4 = ZZ(H4[-1]).digits(2,None,32)
    hh = h4+h3+h2+h1+h0
    return ZZ(hh,2)

print hex(sha1(""))
print hex(sha1("The quick brown fox jumps over the lazy dog"))
```